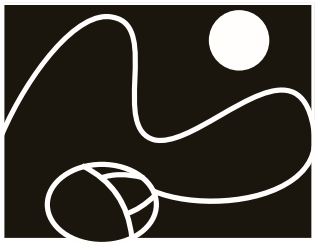
SugarLoafPLoP'2005



5th Latin American Conference on Pattern Languages of Programming

Campos do Jordão, Brazil, August 16-19, 2005

Conference Organization

General Chair: Rosana Teresinha Vaccare Braga, *ICMC-USP, Brazil*

Program co-chairs: Linda Rising, *Independent Consultant, USA*
Fabio Kon, *IME-USP, Brazil*

Local Arrangements co-Chairs: Fabio Kon, *IME-USP, Brazil*
Marcos Cordeiro d'Ornellas, *UFSM, Brazil*
Paulo Cesar Masiero, *ICMC-USP, Brazil*
Rosangela A. D. Penteado, *UFSCAR, Brazil*

Program Committee: Claudia Werner, *UFRJ, Brazil*
Dick Gabriel, *Sun Microsystems, USA*
Eugene Wallingford, *U. Northern Iowa, USA*
Fabio Kon, IME/USP, *Brazil*
Gustavo H. Rossi, *Lifia/UNLP, Argentina*
Jim Coplien, *Vrije Universiteit, Belgium*
Jorge L. Ortega Arjona, *UNAM, Mexico*
Joseph Yoder, *U. Illinois / The Refactory, Inc, USA*
Linda Rising, *Independent Consultant, USA*
Marcos Cordeiro d'Ornellas, *UFSM, Brazil*
Paulo Borba, UFPE, *Brazil*
Paulo Cesar Masiero, *ICMC/USP, Brazil*
Robert Hanmer, *Lucent Technologies, USA*
Rosana Braga, *ICMC/USP, Brazil*
Rossana Andrade, *DC/UFC, Brazil*

Assistant Editor Giuliano Mega, *IME-USP, Brazil*

Edited by:
Linda Rising
Fabio Kon

Conference Proceedings

Supporting Agencies

Sponsors

Organization


IME-USP

Latin American Conference on Pattern Languages of
Programming SugarLoafPLoP 2005 (5.:2005 : Campos do Jordão, SP)
Proceedings... / Editors Linda Rising, Fabio Kon. -- Campos do Jordão,
SP: ICMC/USP 2005
258 p.
ISBN 85-87837-09-5

1. Padrões de Software. 2. Linguagens de Padrões. I. Rising, Linda,
ed. II. Fabio Kon, ed. III. Título

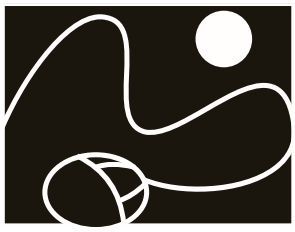
Beneficiário de auxílio financeiro da CAPES - Brasil

Table of Contents

Section 1: Writer's Workshop

Table of Contents

Section 2: Pattern Applications



SugarLoaf PLoP'2005

Writer's Workshop

Padrões para Apoiar o Projeto de Material Instrucional para EAD

Americo Talarico Neto, Junia C. Anacleto, Vânia P. de Almeida Neris

Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – CEP.13565-905 – São Carlos – SP – Brasil

{americo,junia,vania}@dc.ufscar.br

Abstract. *This work presents a pattern collection aiming at supporting teachers to design instructional material for Distance Learning Systems. Such patterns were identified and written based on a selected set of Cognitive Strategies, in order to better organize the content displayed as instructional material presented to students, in an attempt to facilitate their learning process.*

Resumo. *Este trabalho apresenta uma coleção de Padrões com a finalidade de apoiar o professor durante o projeto de material instrucional para sistemas de Educação à Distância (EAD). Tais Padrões foram identificados e escritos a partir de um conjunto de Estratégias Cognitivas, selecionadas com o objetivo de melhor organizar o conteúdo visto pelo aluno, visando facilitar o seu processo de aprendizado.*

1. Introdução

A disseminação da informação, associada ao desenvolvimento das mídias interativas, vem colocando recursos como o computador e a Internet a serviço da educação, e tem gerado uma grande transformação nos processos de ensino e aprendizagem, relacionada principalmente ao uso da Educação à Distância (EAD) como forma de atingir novos públicos e desenvolver novas metodologias de ensino.

O projeto de cursos em ambientes WEB pode ser uma tarefa difícil para os professores que têm pouca experiência em interação e projeto de material instrucional em ambiente computacional. Essa dificuldade acaba gerando uma produção de cursos deficientes que impedem ou dificultam o processo de aprendizagem dos alunos [Frizell, 2001].

Este artigo explora a questão do projeto de material instrucional para EAD, sintetizando algumas propostas disponíveis na literatura da ciência cognitiva, que tenta explicar como ocorre o processo de ensino e aprendizagem no ser humano, expressa aqui em um conjunto adotado de Estratégias Cognitivas [Liebman, 1998], documentando tais práticas em forma de Padrões para apoio adequado ao processo de geração do material instrucional.

Espera-se que tais Padrões possam gerar um vocabulário comum entre os diversos participantes (professores, autores, educadores, profissionais da computação, e web designers) do projeto multidisciplinar de material instrucional para EAD, extraindo e estruturando abstrações de qualidades comuns, identificando soluções e apresentando

a relevância de tais soluções para ajudar os professores a melhor organizar o material instrucional e assim favorecer o aprendizado dos alunos que venham a utilizá-lo.

Neste trabalho utiliza-se o termo professor para designar o profissional responsável pelo projeto de material instrucional para ambiente Web e utiliza-se o termo aluno para designar o usuário que irá interagir com a interface elaborada nesse projeto (o material instrucional).

Este trabalho está organizado da seguinte forma: na seção 2 são apresentadas as Estratégias Cognitivas adotadas como base para este trabalho, na seção 3 são mostradas as principais características dos padrões identificados e na seção 4 os padrões são apresentados em detalhes. Por fim, na seção 5 são feitas as considerações finais.

2. Estratégias Cognitivas para Apoio ao Ensino

Gagné (1974) aborda os processos internos de aprendizagem por meio de itens que foram denominados domínios. Um desses domínios é constituído pelas Estratégias Cognitivas, que segundo ele são capacidades internamente organizadas que o aluno usa para guiar seus próprios processos de atenção, aprendizagem, memória e pensamento. O aluno usa uma Estratégia Cognitiva, por exemplo, ao prestar atenção nas diversas características daquilo que está lendo. O leitor usa certas Estratégias Cognitivas para selecionar e codificar o que aprende, valendo-se de outras estratégias para recuperar posteriormente essas informações [Almeida e Silva, 2004].

As Estratégias Cognitivas são, portanto, os meios que o aluno dispõe para administrar seus próprios processos de aprendizagem. Gagné relaciona tais estratégias com os conceitos de "aprender a aprender" e "aprender a pensar".

Beckman (2002) define as Estratégias Cognitivas como “uma estratégia ou um grupo de estratégias ou procedimentos que os alunos usam para cumprir tarefas acadêmicas ou melhorar habilidades sociais. Normalmente, mais do que uma Estratégia Cognitiva é utilizada, dependendo do esquema de aprendizado do aluno”. As estratégias citadas por Beckman são: Visualização, Verbalização, Associações, Particionamento, Questionamento, Inspeção, Sinalização, Uso de mnemônicos, Auto-verificação e Monitoramento.

Rosenshine (1997) reforça que a melhor maneira de saber que estratégia utilizar é observar como os alunos mais experientes resolvem os problemas e que estratégias utilizam. Algumas das estratégias citadas em seu trabalho são: Quebra de tarefas, Suporte, Feedback e Mapa de Conceitos.

West *et al.* (1991) sugerem o uso de mais algumas Estratégias Cognitivas. As atividades apresentadas por West *et al.* (1991) e utilizadas, com sucesso, por Liebman (1998) no ensino presencial são listadas a seguir: Organização, Estruturação, Mapa de Conceitos, Metáforas e analogias, Ensaios e Organizadores de avanço.

Neste trabalho foram adotadas as estratégias de Liebman (1998). Essa decisão foi tomada após uma análise minuciosa das estratégias anteriormente mencionadas na qual pôde-se perceber que o grupo de estratégias de Liebman reflete quase todas as estratégias citadas anteriormente. Nesse sentido, as estratégias de Liebman são detalhadas a seguir:

- Organização: na literatura sobre psicologia cognitiva é chamada de particionamento, inclui a aplicação de taxonomias, listagem de semelhanças e diferenças, análise de forma e função, listar vantagens e desvantagens e identificar causa e efeito;
- Estruturação: são organizações visuais da estrutura básica da informação em questão; um exemplo de estruturação é a elaboração de uma tabela na qual as linhas representam objetos e as colunas representam as propriedades. O professor fornece a estrutura e pede aos aprendizes que preencham algumas ou todas as informações. Essa estruturação pode ser de dois tipos. No tipo 1 os aprendizes preenchem a estrutura usando a informação que têm disponível, e no tipo 2 eles usam o raciocínio para desenvolver a informação a ser colocada na estrutura;
- Mapa de Conceitos: diagramas usados para expressar relacionamentos temporais, por categoria, causais, hierárquicos, etc;
- Uso de metáforas e analogias;
- Ensaios: são estratégias para manter a informação sendo processada na memória de trabalho dos aprendizes o tempo suficiente para que seja melhor estabelecida na memória de longa duração. Incluem repetição, perguntas e respostas, prever e esclarecer, redefinir ou parafrasear a informação, revisar e resumir, selecionar qual a informação importante, tomar notas e enfatizar (sublinhar);
- Organizadores de avanço: são observações feitas pelo professor para ajudar o aprendiz a passar para um novo tópico, podendo ser entendidos como conectores ou pontes, fazendo associações entre um tópico que está por vir e o conhecimento já adquirido;

Outro ponto interessante que correlaciona este trabalho com o de Liebman é que ela também reconhece que os professores podem utilizar as Estratégias Cognitivas para facilitar o processo de ensino e aprendizagem do aluno. Aqui, neste trabalho, as estratégias são selecionadas e utilizadas pelos professores no projeto do material instrucional com o objetivo de melhor organizar o conteúdo pela interface, em uma tentativa de facilitar o processo de aprendizado do aluno.

3. Características dos Padrões para EAD Identificados neste Trabalho

O objetivo de Alexander na publicação de sua Linguagem de Padrões [Alexander et al., 1977] era permitir aos usuários leigos, os habitantes, a capacidade de participar do projeto de seus ambientes. Essa preocupação é similar às idéias encontradas em Engenharia de Software, no Projeto Centrado no Usuário e no Design Participativo, cujo objetivo é envolver usuários finais em todos os estágios do ciclo de desenvolvimento de software [Borchers, 2001].

Uma Linguagem de Padrões para EAD deve conter Padrões que orientem os instrutores em como elaborar o curso, ajudem na concepção de um projeto para elaborar a seqüência de ações em um curso e forneçam auxílio durante a realização do curso com estratégias de acesso [PPP, 2005].

Tais benefícios estão presentes nesta pesquisa que se propõe a apresentar uma coleção de Padrões para apoiar o professor na tarefa de projetar o material instrucional para ser inserido em sistemas de EAD. Nesse contexto, o professor pode desempenhar os papeis de:

- Usuário: quando atua em conjunto com uma equipe multidisciplinar para o projeto do material instrucional, podendo assim utilizar os Padrões como ferramenta para estabelecer uma comunicação mais eficiente e participar mais ativamente do projeto, utilizando terminologia e conhecendo os problemas do domínio da EAD.
- Projetista: quando ele mesmo projeta e disponibiliza o material instrucional, utilizando os Padrões como ferramenta para avaliação, projeto e validação e para divulgar seus conhecimentos para pessoas menos experientes.

Os Padrões identificados nesta pesquisa foram obtidos por meio de estudos de caso (Almeida e Silva, 2004) cujos objetivos principais foram verificar se o conjunto selecionado de seis Estratégias Cognitivas [Liebman, 1998], apresentado na seção anterior, aumenta a usabilidade de materiais instrucionais para EAD, verificando em que momento essas Estratégias Cognitivas são inseridas no material instrucional e se elas podem ser vistas como soluções para problemas recorrentes nesse contexto e, desse modo, se podem ser escritas na forma de Padrões.

O formato e o estilo de escrita desses Padrões para EAD foram baseados na "Linguagem de Padrões para escrita de Padrões" de Meszaros e Doble (1996), que mostra que os Padrões são mais fáceis de compreender e aplicar quando os elementos **Nome, Forças, Contexto, Problema e Solução** estão presentes no formato utilizado. Esses autores comentam que outros elementos podem ser incluídos no formato do Padrão, mas são opcionais (**Exemplos, Raciocínio**, etc). Segundo esses autores, os elementos opcionais devem ser escolhidos pelo autor para tornar mais fácil a compreensão do Padrão ou para relacionar os Padrões.

Neste trabalho, oito elementos foram considerados como sendo de alta importância para o entendimento dos Padrões formalizados, conforme descrito a seguir: **Nome do Padrão, Contexto, Forças, Problema, Solução, Raciocínio, Exemplos e Padrões Relacionados.** Cada um dos Padrões identificados foi escrito seguindo-se essa estrutura.

Os relacionamentos entre os Padrões da coleção obtida são mostrados na Figura 1, construída baseada na teoria de Linguagem de Padrões de Alexander *et al.* (1977), que relaciona os Padrões visualmente na forma de um grafo, no qual as caixas (nós) representam os Padrões e as linhas (arestas) representam os relacionamentos entre eles.

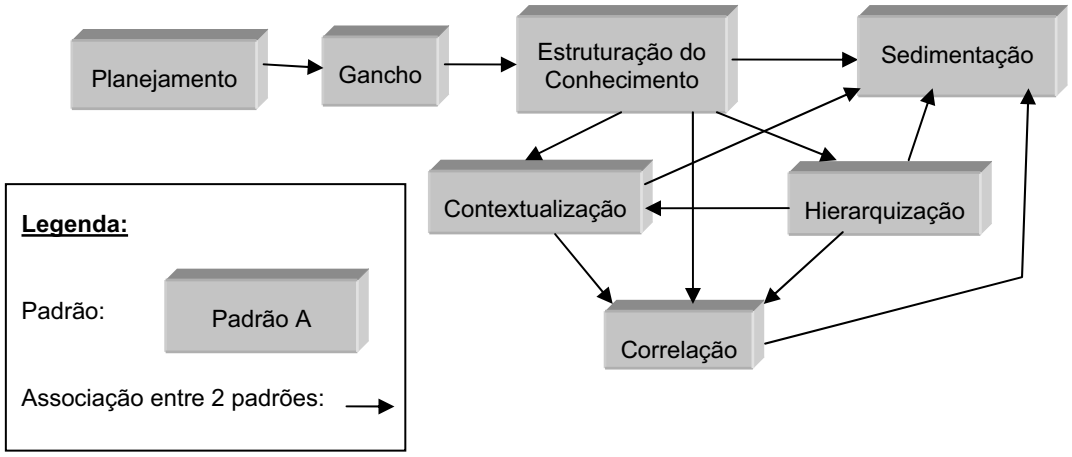


Figura 1. Relacionamentos entre os Padrões para EAD, identificados neste trabalho.

A seguir serão apresentados todos os Padrões que foram obtidos a partir das Estratégias Cognitivas apresentadas na seção 2.

4. Padrões Identificados a partir das Estratégias Cognitivas

4.1. O Padrão Planejamento

Contexto:

A primeira tarefa do professor é planejar a aula estruturando os conteúdos e criando um ambiente que motive o aluno. Esta fase define a fundação necessária para conduzir uma boa aula.

Forças:

- A preparação de uma aula completa envolve o entendimento de uma série de conceitos e interesses, bem como o entendimento de que os diferentes públicos têm habilidades e conhecimentos únicos.
- O professor geralmente tem familiaridade com o tema da aula, entretanto ele pode esquecer de mencionar tópicos que são importantes para o entendimento do tema pelo aluno.
- O Planejamento auxilia o professor a melhor organizar uma aula e facilita o processo de transferência do conhecimento.

Problema:

Como planejar a transferência de conhecimento do professor para o aluno?

Solução:

Formalize o problema a ser resolvido, definindo o objetivo final, que o ajudará a determinar as estratégias para uma aula. Especifique um ou mais sub objetivos que seu material instrucional deve contemplar e que aspectos você quer focalizar.

Identifique o conhecimento inicial necessário que o aluno deve ter e siga os seguintes passos:

1- Definição dos resultados de aprendizagem desejados (quando esses resultados forem muito complexos, dividi-los em resultados mais simples)

2- Estabelecimento de uma hierarquia de resultados,

3- Identificação das condições internas requeridas,

4- Identificação das condições externas requeridas,

5- Planejamento dos meios de aprendizagem em função do contexto de aprendizagem e das características do grupo,

6- Planejamento da motivação

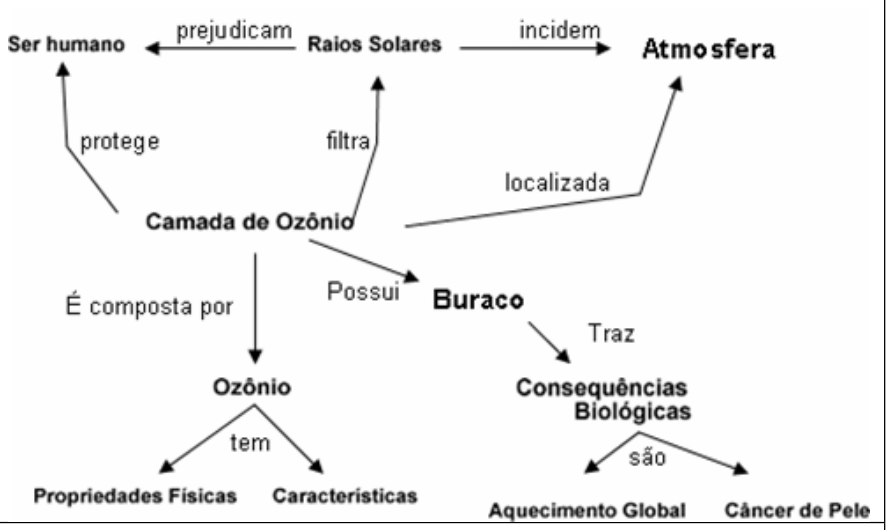
Raciocínio:

Um bom planejamento é resultado de experiência. Esteja atento para adaptar o planejamento para próximas versões da mesma aula, incorporando assim novas experiências.

Exemplos:

Considere o projeto de um material instrucional que aborda o tema "A Camada de Ozônio". O professor estabelece o seguinte plano de ensino com objetivos de aprendizado para seus alunos, durante o contato com o material instrucional. A partir do plano de Ensino o professor elabora um Mapa de Conceitos para organizar as idéias e conceitos que ele gostaria de transmitir à seus alunos, na forma de conteúdo:

Plano de Ensino	
Identificação	Aula: "A Camada de Ozônio"
Disciplina:	Biologia – Meio ambiente
Pré-requisitos	Conhecer os conceitos de Meio Ambiente, atmosfera e as condições para preservação das formas de vida na Terra.
Ementa:	Apresentação da Camada de Ozônio, sua composição, localização e os problemas biológicos causados pela sua destruição
Objetivos:	Como a camada de ozônio se forma? Por que ela é importante? Como a camada de ozônio vem sendo destruída? Quais as conseqüências biológicas dessa destruição?

Mapa de Conceito	
Referências	Enciclopédia Britânica

Padrões Relacionados: Gancho.

4.2. O Padrão Gancho

Contexto:

O professor realizou o planejamento da sua aula e já tem os objetivos de ensino e aprendizado bem consolidados. Agora é necessário dar início ao que foi planejado, mostrando ao aluno o que ele irá aprender.

No início de uma aula é interessante mostrar ao aluno o conceito principal do assunto que ele irá aprender e se tal conceito é relacionado com algum outro previamente conhecido, para que o aluno possa estabelecer relacionamentos entre tais conceitos.

Para preparar o aluno para a integração do conhecimento é necessário construir uma ponte entre o material novo e as idéias existentes.

Forças:

- A introdução de um novo conceito pode fazer com que o aluno se sinta desorientado durante o início de uma aula virtual e consequentemente tenha seu aprendizado dificultado. Ao estimular o aluno a relembrar conceitos que ele já domina e a relacioná-lo com o conceito que será apresentado, o professor atua como facilitador do aprendizado do aluno.
- Na aula, ao final da apresentação de um conceito, este pode ser usado como uma introdução ao próximo conceito a ser aprendido, preparando o aluno para receber um novo tema com base em um conceito já assimilado.

Problema:

Como o professor pode apresentar uma nova aula ao aluno?

Solução:

Utilize a Estratégia Cognitiva Organizadores de Avanço.

Um organizador de avanço é uma Estratégia Cognitiva proposta por David Ausubel (1968) que serve como tópico ou categoria nos quais os fatos e os detalhes podem ser organizados e subseqüentemente aprendidos. Os organizadores de avanço são importantes para auxiliar o aluno a aprender, recordar, e relacionar o material que já estudou. Podem incluir observações feitas pelo professor para ajudar o aluno a iniciar um novo tópico.

Apresente material introdutório que ajude o aluno a relacionar informação nova com esquemas de conhecimentos existentes. Novas idéias e conceitos devem ser potencialmente significativos para o aluno. Ajude-o a relacionar novas idéias com conhecimento existente.

Estimule o aluno a responder perguntas tais como:

- O que você quer descobrir?
- Que ações você deve fazer para chegar lá?
- O que você já sabe?

Raciocínio:

Os Organizadores de avanço, propostos por David Ausubel (1968), ajudam a construir uma fundação. Uma inspeção prévia do material a ser estudado e aprendido forma uma estrutura de conhecimento prévio sobre os quais o conhecimento novo e a compreensão podem ser construídos.

Os tipos de Organizadores de Avanço são:

Organizadores de Avanço Expositores: pode simplesmente fornecer aos alunos o significado e a finalidade do que deve seguir. Por outro lado, pode apresentar aos alunos informação mais detalhada do que estarão aprendendo especialmente a informação que pode ser difícil de compreender.

Organizadores de Avanço Narrativos: tem o formato de uma história. Aqui o professor fornece as idéias essenciais de uma aula ou de uma unidade que planeja ensinar contando uma história que incorpore as idéias.

Organizador de avanço superficial: o professor fornece aos alunos uma oportunidade de inspecionar a informação importante que encontrará mais tarde focalizando os títulos, os subtítulos, e as informações destacadas. Utilize a Estratégia Cognitiva Ensaios para selecionar a informação importante, sublinhar ou destacar. Ensaio é definido como atividades que ajudam processar o material na memória de curta duração deixando-o ativo na consciência do aluno para que ele possa ser recuperado mais tarde [Mayer, 1987].

Organizadores gráficos: fornecem aos alunos a orientação de qual informação importante uma lição ou uma unidade é composta. Dão a alunos o sentido e fornecem também uma representação visual da informação importante.

Exemplos:

No início de uma aula sobre “a Camada de Ozônio” é apresentado um Organizador de Avanço Narrativo para explicar o fato de que a Terra possui um escudo que filtra os raios solares que são maléficos aos seres humanos, estimulando o aluno a relembrar os

conceitos de atmosfera e raios ultravioletas e que esse filtro é composto pelo gás ozônio, que vem sendo destruído ultimamente.

Figura 2. Exemplo da Estratégia Cognitiva Organizador de Avanço.

Padrões Relacionados: Estruturação do Conhecimento

4.3. O Padrão Estruturação do Conhecimento

Contexto:

O professor estimulou o conhecimento prévio do aluno que agora sabe como o tópico que ele irá aprender se relaciona com conceitos que ele já conhece. Agora o professor deve mostrar os conceitos principais, bem como o conteúdo que deve ser aprendido.

Forças:

- Para manter o estudante ativo durante uma aula virtual, o professor pode mostrar como o conceito que vai ser apresentado será explorado e detalhado, pois os alunos geralmente se lembram melhor do que eles aprendem inicialmente e têm a necessidade de saber o tamanho da aula, seus tópicos principais e o progresso.
- O professor pode introduzir as idéias importantes no início da aula, mesmo que elas não sejam completamente exploradas de imediato. Dessa forma o aluno terá uma visão geral do conhecimento que ele irá aprender.

Problema:

Como podemos introduzir novos conceitos aos alunos?

Solução:

Utilize Mapa de Conceitos como ferramenta para a indexação dos conteúdos envolvidos em um ambiente virtual de aprendizagem. O ambiente deve conter uma página para

cada nó (Conceito) do Mapa de Conceitos e um índice, que serve como "link" para elas. Os Mapa de Conceitos são úteis por diversas razões: são um registro observável da compreensão de um indivíduo; demonstram como a informação é significativa; forçam um indivíduo a pensar sobre seus próprios processos de pensamento e estruturação do conhecimento.

Mapa de Conceitos podem ser utilizados como ferramenta instrucional para:

- Organizar o índice do curso: construindo um mapa de todas as idéias de um curso, os professores podem usar tal estrutura para organizar o índice do curso. Isto fornece uma maneira para o instrutor ver conexões entre o material do curso e como melhor apresentar as conexões aos alunos
- Preparar aulas específicas: melhor que mapear o índice de um curso inteiro, um instrutor pode focalizar na tarefa mais específica de traçar o índice de somente uma aula para questões de melhor organização.
- Apresentar o material aos alunos: um instrutor pode escolher ensinar o material do curso com o uso de Mapa de Conceitos para mostrar claramente as conexões entre conceitos.

Utilize a Estratégia Cognitiva Ensaios para selecionar informações importantes, para facilitar a localização dos itens e sua identificação no texto. Ensaio é definido como atividades que ajudam a processar o material na memória de curta duração, deixando-o ativo na consciência do aluno para que ele possa ser recuperado mais tarde [Mayer, 1987].

Utilize a Estratégia Cognitiva Estruturação (na forma de listas), que são organizações visuais da estrutura básica da informação em questão.

Raciocínio:

A técnica de Mapa de Conceitos, desenvolvida pelo Prof. Joseph D. Novak (1977), está embasada na teoria construtivista, entendendo que o indivíduo constrói seu conhecimento e seus significados a partir da sua predisposição para realizar essa construção, e servem como instrumentos para facilitar o aprendizado do conteúdo sistematizado em conteúdo significativo para o aprendiz.

Exemplos:

A figura abaixo apresenta um exemplo de um índice para a aula sobre “a Camada de Ozônio”, baseado na Estratégia Cognitiva Mapa de Conceito utilizada como indexador de conteúdos durante o Planejamento da Aula (veja padrão Planejamento). O projeto navegacional do material instrucional foi feito de tal forma que cada página Web desse material representasse um conjunto de conhecimento (tópicos) que se queria transmitir.

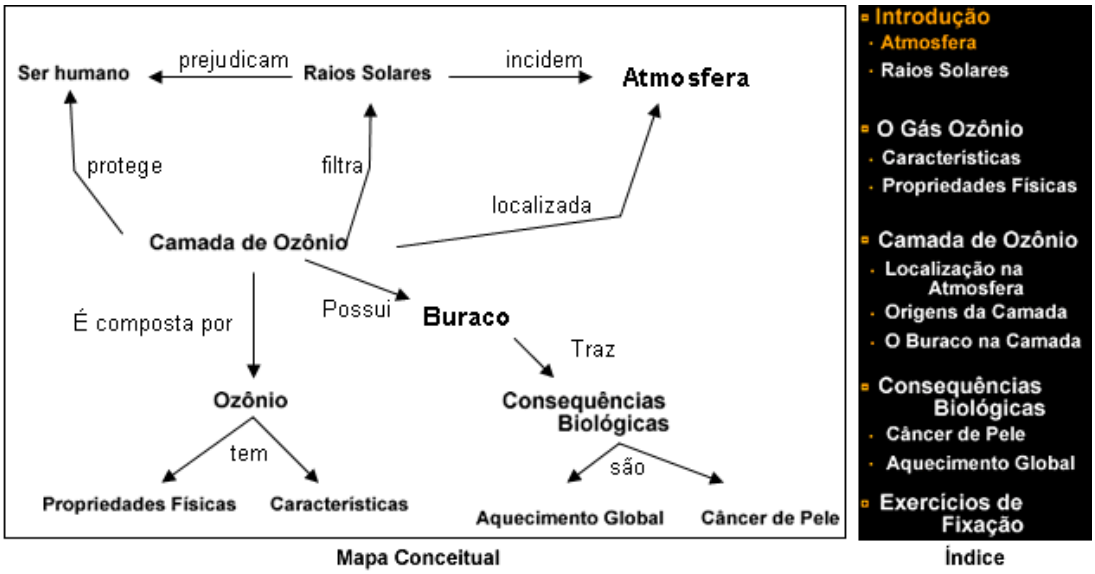


Figura 3. Exemplo da Estratégia Cognitiva Mapa de Conceitos utilizado como indexador de conteúdos.

Padrões Relacionados: Correlação, Contextualização, Hierarquização, Sedimentação.

4.4. O Padrão Contextualização

Contexto:

Os alunos estão estudando o material instrucional e em certo ponto gostariam de saber como os conceitos que estão aprendendo se relacionam com o ambiente em que vivem e como podem aplicá-lo**.**

Forças:

- Para manter o aluno mais engajado em uma aula virtual é necessário fazê-lo visualizar como o conceito que está aprendendo pode ser aplicado no seu dia a dia, ou em seu ambiente de trabalho e as motivações para que ele possa utilizar tais conceitos para resolver seus problemas.

Problema:

Como aplicar o conceito recentemente mostrado ao ambiente do aluno?

Solução:

O conteúdo não deve ser apresentado apenas de forma expositiva e descritiva. Sempre que possível, o tema deve ser introduzido por alguma atividade em que se resgatem os conhecimentos e as informações que o aluno traz, criando-se, assim, um contexto que irá dar um "significado" ao tema em questão, justificando ainda o fato do por que tal tema está sendo estudado.

Utilize a Estratégia Cognitiva Ensaios para selecionar a informação importante, para facilitar a localização dos itens e sua identificação no texto. Ensaio é definido como atividades que ajudam processar o material na memória de curta duração deixando-o

ativo na consciência do aluno para que ele possa ser recuperado mais tarde [Mayer, 1987].

Utilize a Estratégia Cognitiva Organizações que na psicologia cognitiva é também conhecida como particionamento e sugere a aplicação de taxonomias, listagem de semelhanças e diferenças, análise de forma e função, listar vantagens e desvantagens e identificar causa e efeito.

Utilize a Estratégia Cognitiva Mapa de Conceitos para expressar relacionamentos entre os conceitos apresentados em forma de diagrama

Exemplos:

A figura abaixo apresenta parte de um Material Instrucional sobre a aula "A Camada de Ozônio" elaborado com as Estratégias Cognitivas: Ensaios para selecionar a informação importante e facilitar a sua localização e identificação no texto e Estruturação para identificar as causas e os efeitos da exposição moderada e excessiva ao Sol. Tais conceitos são úteis para o aluno e a forma como são apresentados reforça a sua importância.

Figura 4. Material Instrucional elaborado com as Estratégias Cognitivas Estruturação e Ensaios.

Padrões relacionados: Correlação, Sedimentação

4.5. O Padrão Hierarquização

Contexto:

Os tópicos em um curso são divididos em fragmentos e os fragmentos são introduzidos em uma ordem que facilite resolver um problema do aluno. Muitos dos fragmentos introduzem um conceito, mas não o cobrem em detalhes. Inicialmente, o tratamento dado é suficiente apenas para formação de uma compreensão básica dos conceitos que serão reforçados e detalhados posteriormente em seqüências adicionais.

Forças:

- É necessário que o aluno conheça todos os tópicos que ele irá estudar antes de aprender cada conceito individualmente, pois o cérebro aprende melhor quando ele consegue associar novos assuntos com assuntos aprendidos e quanto mais associações forem feitas pelo cérebro, mais fácil será recuperar o conhecimento adquirido e aplicá-lo em certo ambiente.
- Tópicos extensos, requerem muitos fragmentos com diversos conceitos envolvidos. O material instrucional necessário para explicar todos os conceitos envolvidos pode ser facilmente expandido em subitens causando poluição textual. É necessário que os alunos saibam de antemão quais são os conceitos importantes antes de saber suas explicações.

Problema:

Como podemos introduzir um conceito que tem um grande número de subitens?

Solução:

As idéias mais gerais de um assunto devem ser apresentadas primeiramente e depois progressivamente diferenciadas em termos de detalhes. Organize o novo material por coordenação, subordinação e superordenação.

Segundo a idéia de diferenciação progressiva, se o objetivo é ensinar os itens X, Y e Z, deve-se, primeiro, ensinar os 3 itens num nível geral, depois os 3 itens num nível de maior detalhe e assim por diante; o oposto seria ensinar tudo sobre X, depois tudo sobre Y e depois tudo sobre Z. De início, serão apresentadas as idéias mais gerais que serão, progressivamente, detalhadas em termos de detalhe e especificidade. Importante nesse processo é, a cada passo, destacar o que os itens têm em comum e o que os diferencia.

Utilize a Estratégia Cognitiva Ensaios para selecionar a informação importante, para facilitar a localização dos itens e sua identificação no texto..

Utilize a Estratégia Cognitiva Estruturação, que são organizações visuais da estrutura básica da informação em questão.

Raciocínio:

A diferenciação progressiva vê a aprendizagem significativa como um processo contínuo no qual adquirem significados mais abrangentes à medida que são estabelecidas novas relações entre os conceitos.

Exemplos:

A figura abaixo apresenta uma página de um material instrucional sobre “A Camada de Ozônio” elaborado utilizando-se a diferenciação progressiva, na qual todos os itens são apresentados antes de suas explicações para se ter uma visão geral do que será aprendido. A Estratégia Cognitiva Estruturação foi usada para organizar visualmente a informação em questão.

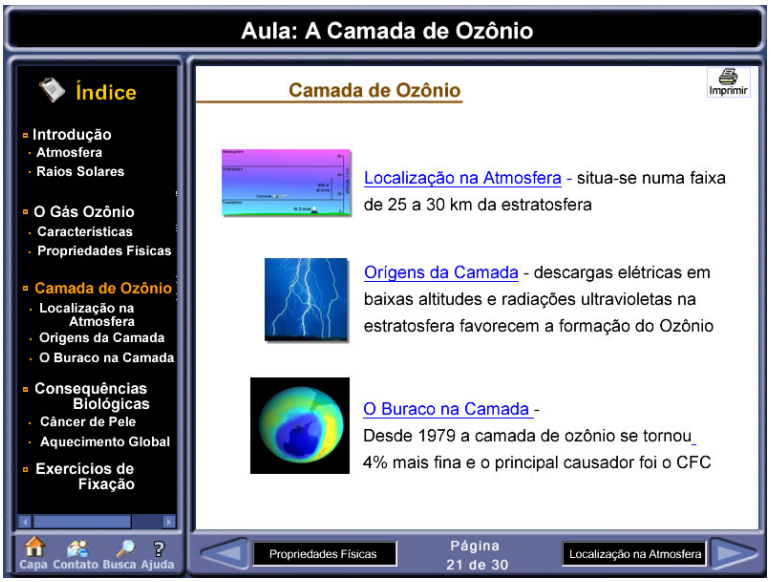


Figura 5. Exemplo de material Instrucional elaborado com Diferenciação Progressiva.

Padrões relacionados: Correlação, Contextualização, Sedimentação

4.6. O Padrão Correlação

Contexto:

Ao ensinar um tópico complexo fora da experiência normal do aluno, encontre uma metáfora complexa e consistente para o tópico que está sendo ensinado. O contexto base da metáfora necessita ser de conhecimento dos alunos.

Forças:

- Os alunos precisam de uma estratégia poderosa e consistente para pensar sobre algum tópico complexo. A estratégia deve relacionar o tópico que está sendo ensinado ao contexto que o aluno vivencia.
- Os alunos podem ficar perdidos nos detalhes facilmente e podem não ver como as peças se relacionam. Isto é válido quando os detalhes são estranhos ou novos aos alunos.

Problema:

Como fazer com que os alunos vejam rapidamente como o tópico se relaciona com os objetivos maiores da aula e entendam como os conceitos se relacionam?

Solução:

Utilize a Estratégia Cognitiva Metáforas e Analogias. Crie uma Metáfora que seja consistente com o tópico que está sendo ensinado. Forneça aos alunos uma maneira rápida de pensar sobre o tópico.

Exemplos:

Durante uma aula sobre “A camada de Ozônio” o professor cria uma analogia entre a Camada de Ozônio e um escudo que protege a Terra dos raios nocivos do Sol, com o objetivo de trabalhar a informação de maneira diferente na memória do aluno. Durante a

mesma aula é apresentada uma metáfora para relacionar a destruição da camada de ozônio com o lançamento de gases tóxicos na atmosfera.

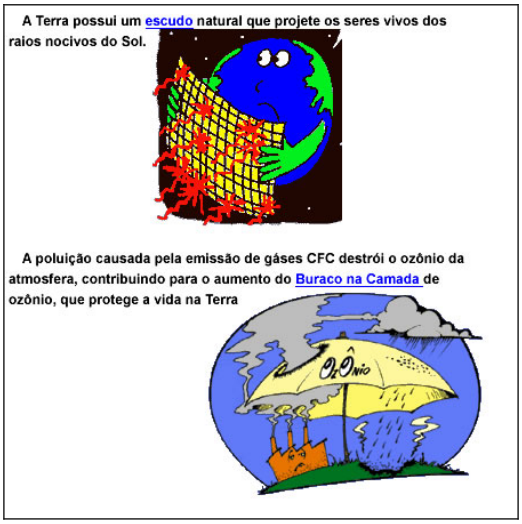


Figura 6. Exemplo de Metáforas e Analogias inseridas no material instrucional.

Padrões relacionados: Sedimentação.

4.7. O Padrão Sedimentação

Contexto:

O aluno estudou uma quantidade razoável do material instrucional e precisa que essa informação seja trabalhada por mais tempo em sua memória, enquanto ele se prepara para adquirir novos conhecimentos.

Forças:

- O cérebro consegue se concentrar em um determinado tópico por um período limitado. Após esse período os alunos não conseguem aprender eficientemente.
- É preciso manter o conhecimento novo na memória do aluno e fazer com que ele estabeleça relações com o que já conhece, bem como exercitá-lo em problemas reais.

Problema:

Como fazer com que o novo conhecimento adquirido fique sendo trabalhado na memória de curta duração do aluno, enquanto ele se prepara para adquirir novos conhecimentos?

Solução:

Integre o novo conhecimento a outras áreas de conhecimento.

Os materiais instrucionais devem tentar integrar o material novo com informação previamente apresentada por meio de comparações que referenciem idéias novas e velhas, considerações, tabelas, conclusão e exercícios.

Para facilitar esse processo, o material instrucional deve procurar integrar qualquer material novo com material anteriormente apresentado (referências, comparações etc.),

inclusive com exercícios que exijam o uso do conhecimento de maneira nova (por ex: formulação de questões de maneira não familiar).

Raciocínio:

Reconciliação Integradora é o processo pelo qual a pessoa reconhece novas relações entre conceitos até então vistos de forma isolada.

Exemplos:

A figura abaixo apresenta parte do material instrucional sobre "A Camada de Ozônio", cuja Retenção de Conhecimento foi elaborada com um exercício do tipo "selecione", com o objetivo de manter o conhecimento recentemente adquirido na memória de longa duração do aluno e fazer com que ele estabeleça relações com o que já conhece.

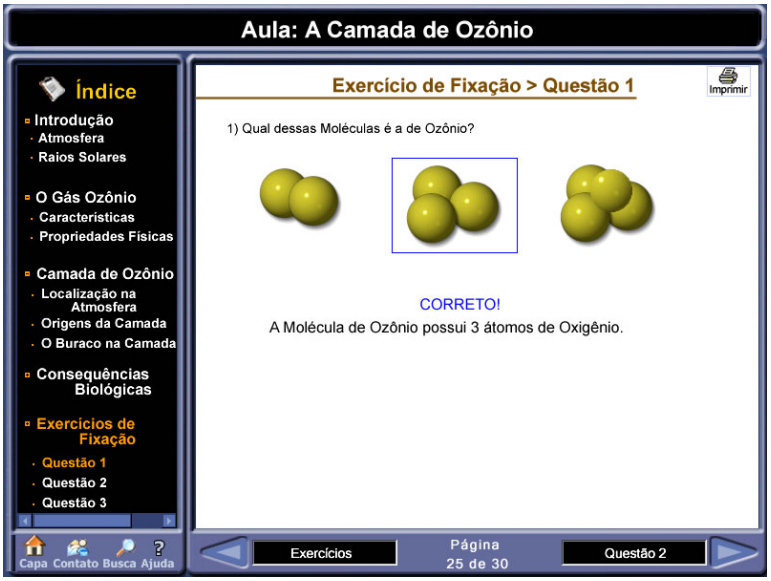


Figure 7. Exercício do tipo "selecione" projetado para realizar a Retenção de Conhecimento

5. Considerações Finais

A coleção de Padrões para EAD proposta neste trabalho foi decorrência do estudo da aplicação das Estratégias Cognitivas utilizadas por Liebman (1998) como uma forma de apoio aos professores na tarefa de projetar material instrucional para EAD com qualidade. Os estudos de caso realizados [Almeida e Silva, 2004] proporcionaram a identificação e escrita dos Padrões para EAD apresentados aqui e permitiram verificar que as Estratégias Cognitivas aumentam a usabilidade do material instrucional para EAD e, consequentemente, sua qualidade.

Como trabalho futuro, espera-se organizar a coleção de Padrões para EAD obtida nesta pesquisa, visando a criação de uma Linguagem de Padrões para EAD que capture princípios pedagógicos e boas práticas de projeto de interação, abordando questões relativas ao projeto do *layout* e utilização de multimídia de uma forma que englobe um maior número de problemas encontrados pelos professores durante o projeto e a geração de materiais instrucionais para ambientes Web.

6. Agradecimentos

Agradecemos ao apoio recebido do projeto TIDIA-Ae da FAPESP (processo 03/08276-3). Agradecemos ao nosso shepherd, o professor Paulo Cesar Masiero, pela valiosa contribuição na melhoria dos padrões apresentados nesse trabalho.

Referências

Alexander, C. et al. "A Pattern Language". Oxford University Press, N.Y., 1977.

Almeida, V. P.; Silva, J. C. A. (2004) Estratégias Cognitivas para Aumento da Qualidade do Hiperdocumento que Contém o Material Instrucional para EAD. In: IHC 2004 - VI Simpósio sobre Fatores Humanos em Sistemas Computacionais. 17-20 de Outubro de 2004. Curitiba

Ausubel, David P. (1968). Educational Psychology, a Cognitive View. New York: Holt, Rinehart and Winston, Inc.

Beckman, P. Strategy Instruction. ERIC Clearinghouse on Disabilities and Gifted Education. http://ericec.org/digests/e638.html

Borchers, J. A Pattern Approach to Interaction Design. John Wiley & Sons Ltd, 2001.

Frizell, Sherri S. "A Pattern-Based Design Methodology for Web-based Instruction". Thesis Research. Auburn University, September, 2001.

Gagné, R. M. The Conditions of Learning. 3rd editon. Holt, Rinehart e Winston, 1974.

Liebman, J. Teaching Operations Research: Lessons from Cognitive Psychology. Interfaces, 28 (2), 1998. 104-110.

Meszaros G. and Doble J. (1996) "MetaPatterns: A Pattern Language for Writing Patterns", in Proceedings of the Conference on Pattern Languages of Programming PloP 1996, Allerton Park, Illinois, Sept. 4-6, 1996, http://www.hillside.net/patterns/writing/patternwritingpaper.htm.

Mayer, R.E. (1987). Educational Psychology: A cognitive approach. Boston: Little, Brown.

Novak, J. D. (1977). A Theory of Education. Ithaca, NY: Cornell University Press.

PPP. Pedagogical Patterns Project. Website visited in 30/01/2005 http://www.pedagogicalpatterns.org.

Rosenshine, B. (1997) The Case for Explicit, Teacher led, Cognitive Strategy Instruction. Annual Meeting of the American Educational Research Association. Chicago. http://www.epaa.asu.edu/barak/barak1.html

A Pattern Language for Adaptive Distributed Systems

Francisco José da Silva e Silva[1], Fabio Kon[2], Joseph Yoder[3], Ralph Johnson[3]

[1]Department of Informatics - Federal University of Maranhão
[2]Department of Computer Science - University of São Paulo
[3]Department of Computer Science - University of Illinois at Urbana-Champaign

fssilva@deinf.ufma.br, kon@ime.usp.br,
joe@joeyoder.com, johnson@cs.uiuc.edu

Introduction

Modern computing environments are characterized by a high level of dynamism. Two major kinds of dynamic changes occur frequently. The first refers to structural changes such as hardware and software upgrades, protocol and API updates, and operating system evolution. The second refers to dynamic changes in the availability of memory, CPU, network bandwidth and, in mobile systems, connectivity and location. Drastic changes may occur in a few seconds, impacting the performance of user applications profoundly. Among existing production software systems few offer support for managing, adapting, and reacting to these changes; most of the times, all the work is left to users and system administrators who must take care of them manually.

Fortunately, this scenario is gradually changing as researchers in academia and industry investigate elegant and robust ways to build self-adaptive systems for the dynamic, distributed environments of the future. In this paper we present a pattern language that captures some of the most relevant problems and solutions faced by developers who accept the challenge of building automatically configurable and adaptive distributed systems.

Dynamic Reconfiguration

Services must grow to meet increasing usage, new requirements, and new applications. However, flexibility usually conflicts with availability. In conventional systems, the service provider must often shut down, reconfigure, and restart the service to update or reconfigure it. In many cases, it is unacceptable to disrupt the services for any period of time. Disruption may result in business loss, as in the case of electronic commerce, or it may put lives in danger, as in the case of mission critical systems delivering disaster information, for example. Research in dynamic reconfiguration seeks solutions to this problem.

By breaking a complex system into smaller components and by allowing the dynamic replacement and reconfiguration of individual components with minimal disruption of system execution, it is possible to combine high degrees of flexibility and availability.

Self-Adaptation

Highly heterogeneous platforms and varying resource availability motivates the need for self-adapting software. Applications can improve their performance by using different

algorithms in different situations and switching from one algorithm to another according to environmental conditions. Significant variations in resource availability should trigger architectural reconfigurations, component replacements, and changes in the components' internal parameters.

Consider, for example, the network connectivity of a mobile computer as its user commutes from work to home. As the user switches from a wired connection at the office, to a wireless WAN using a cellular phone, and finally, to a modem connection at home, the available bandwidth changes by several orders of magnitude. The movement is also accompanied by changes in latency, error rates, connectivity, protocols, and cost.

Ideally, we would like to have a system capable of maintaining an explicit representation of the dependencies among the network drivers, transport protocols, communication services, and the application components that use them. Only then, would it be possible to inform the interested parties when significant changes occur. Upon receiving the change notifications, applications and services would be able to select different mechanisms, replace components, and modify their internal configuration to adapt to the changes, optimizing performance.

Designing Self-Adaptive Systems

The design of a self-adaptive system must answer three key questions:

1. **When to adapt?** How can the system detect that it is time to adapt (change its behavior) so that its performance will improve or that changes in the environment will not harm system correct functioning.
2. **What do adapt?** Which parts, elements, components of the system (e.g., mechanisms, algorithms and protocols) are subject to being adapted or replaced?
3. **How to adapt?** What are the mechanisms that allow for change in behavior? Given a certain system and environmental state, which adaptations would be more beneficial?

Only by addressing the three key questions above, a software framework can provide a comprehensive solution to the problem of building effective self-adaptive systems. In the remaining of this paper we present a pattern language that addresses the most important aspects of dynamic reconfiguration and adaptation in distributed systems.

Note, however, that there are other important non-functional aspects that are orthogonal to the patterns in this language. Aspects such as Security, Fault-Tolerance, and Real-Time can be essential factors for consideration depending upon the different environments the system will be deployed. In these cases, the reader should refer to patterns specific to these domains.

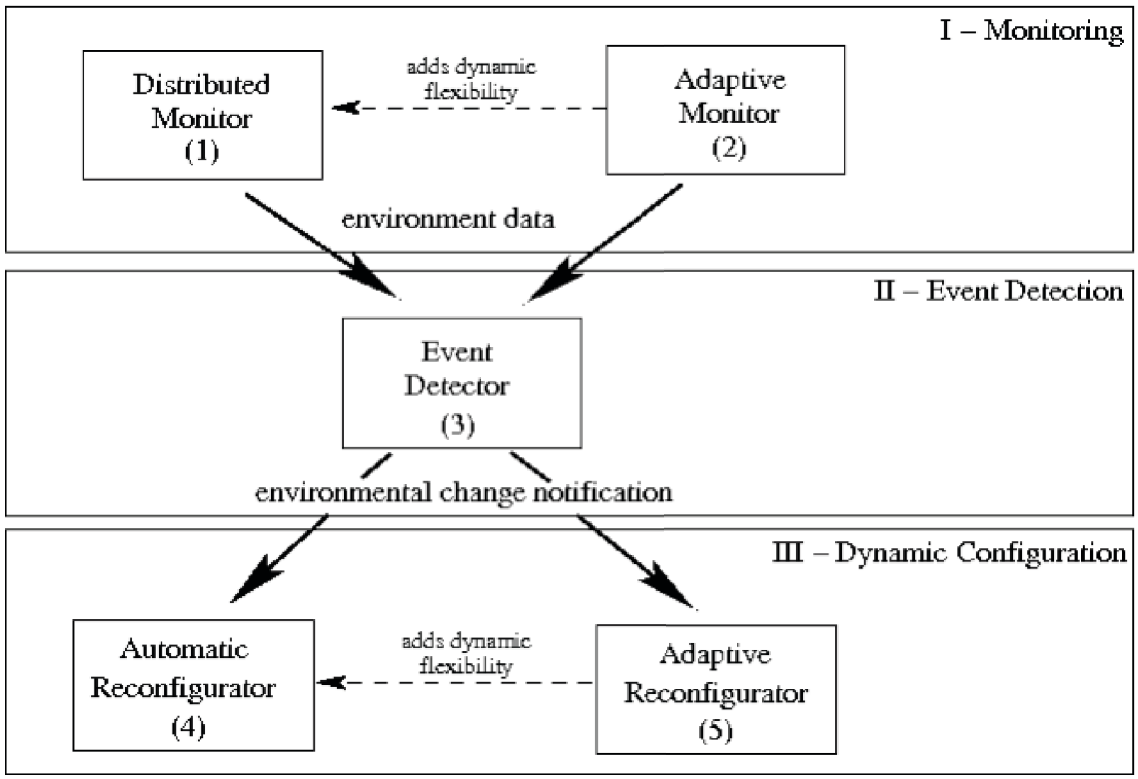


Figure 1: Pattern Language Structure

Figure 1 illustrates the pattern language structure, which is composed of three parts: monitoring, event detection, and dynamic configuration. Part I helps answer the "**When**" question. These patterns are related to monitoring distributed environments. The **DISTRIBUTED MONITOR (1)** and the **ADAPTIVE MONITOR (2)** patterns describe monitoring solutions for applications that must obtain a global view of the state of distributed resources to decide when adaptations should be performed. The **DISTRIBUTED MONITOR (1)** provides a simpler solution while the **ADAPTIVE MONITOR (2)** describes an extension that supports dynamic reconfiguration of the monitor. Both monitors provide monitoring data to event detection mechanisms. Part II presents the **EVENT DETECTOR (3)**, which helps detect when an adaptation should be performed and decides "**What**" adaptations should be performed. When the need for an adaptation is detected (with the information provided by the monitors), the **EVENT DETECTOR (3)** notifies the mechanisms responsible for the dynamic reconfiguration of the system.

Part III shows "**How**" adaptation can be performed with the **AUTOMATIC RECONFIGURATOR (4)** and the **ADAPTIVE RECONFIGURATOR (5)** patterns. Again, the former pattern describes a simpler solution while the latter describes an extension that supports dynamic reconfiguration of the reconfiguration process, leading to a "reconfigurable reconfigurator".

These patterns work together to solve the problem of describing what resources we are concerned about, when to adapt and how to adapt. We now present the five patterns of the pattern language. The patterns are presented in the same order in which information in the system flows, i.e., the monitors collect information, passing it to the detector, which, in its turn, notifies the reconfigurators.

1. DISTRIBUTED MONITOR

Motivation:

In order to improve its performance by means of dynamic configuration, a system must be aware of the dynamic state of the environment in which it is executing. In a distributed system, the environment is spread throughout a collection of, possibly heterogeneous, machines linked by, possibly heterogeneous, network links. Monitoring resource availability can help detect when the system reaches a state in which dynamic reconfiguration would improve application performance or avoid its breakage

Problem:

How to monitor the resources of a distributed system efficiently?

Forces:

- The state of a distributed system is composed of many variables distributed across many machines in various locations. The communication delays and relative speeds of computations of asynchronous distributed systems make it difficult to detect a global state in which dynamic adaptation would be desirable.
- Trying to detect opportunities for dynamic adaptation by looking at isolated machines is easier but this approach cannot provide optimal solutions; it is very likely that relevant global information will be missing.
- Having a single centralized node be aware of the state of the entire system might be infeasible since this compromises scalability (the central node becomes a bottleneck as the system grows) and fault-tolerance (the central node becomes a single point of failure). One can approximate a centralized view of the global state by sending messages from all the nodes to the central node at a high rate. But this may impose an extremely high communication cost and make the central node a single point of failure.
- On the other hand, replicating the central node to avoid a single point of failure increases system complexity and network usage.
- Adopting a lazy protocol in which the state of individual nodes is sent to the central server at a slow rate could solve the network congestion problem but would probably make the data in the central node stale and therefore of little use.

Solution:

Organize the distributed system as a hierarchy of clusters, as illustrated in Figure 2, so that each cluster includes the machines in a local area network, typically containing from a few to approximately one hundred machines (this number can vary depending upon requirements and the environment). Provide a single (possibly replicated) Monitoring Server for each cluster. Have the cluster nodes send periodic information about their local state to the Monitoring Server, as illustrated in Figure 3. These messages may be sent by multicast to all replicated copies of the Monitoring Server (e.g., using the IP-Multicast protocol) so that the network load is not increased by an increase in the number of replicas.

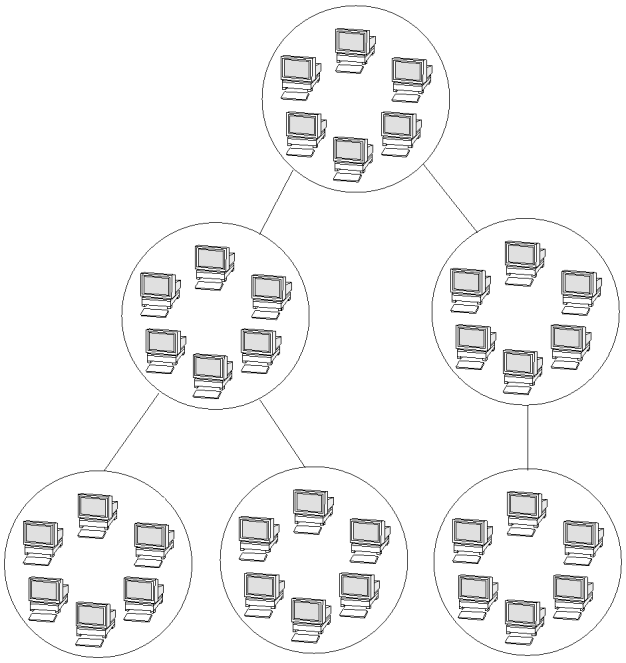


Figure 2 – Hierarchy of computer clusters

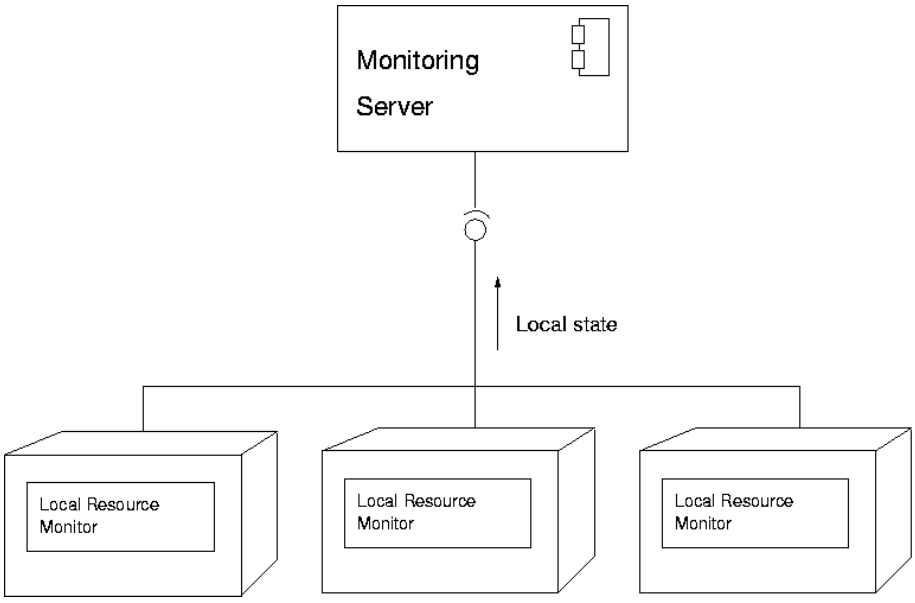


Figure 3 – Distributed Monitoring within a single cluster of machines

To avoid unnecessary messages in the network, the frequency with which the messages are sent to the Monitoring Server cannot be high. Thus, have each node monitor its resources locally at a higher rate (e.g., once per minute) and only send update messages to the Monitoring Server when a significant change in the local state occurs. If no changes occur during a long period (e.g., 5 minutes) then send an update message to the Monitoring Server as a keep-alive. If it is not important whether the nodes are alive or not, then this keep-alive message is not necessary.

The Monitoring Servers of different clusters may be organized in a hierarchy so that consolidated information about cluster state can be exchanged across clusters in a lazy fashion. The farther a Monitoring Server is from a certain machine, the less accurate this monitoring information will be for this machine since changes might have happened to the machine by the time the Server processes the information.

Example:

A distributed system is composed of several distributed resources, such as CPUs, memory, disks, and network links. For each resource, one would like to know the current usage level. Resource usage can be expressed by several properties. For a network link, for instance, properties could be available bandwidth, current latency, number of collisions, etc. In such a system, it would be desirable to have a load balancing mechanism that would migrate tasks from one machine to another depending on resource availability on the distributed system.

Consequences:

+ Network usage is limited (can be fine tuned through the periodicities).
+ One can have a good approximation of the system global state.
_ Implementation in the hierarchical case can be complex, compared to a single centralized server.
_ To implement the monitoring service component that collects the state of local resources in each node one must define, in advance, which machine resources will be monitored.

Resulting Context:

By applying this pattern, a collection of machines, possibly organized in a hierarchy of machine clusters, can be monitored with low network and processor overheads. Thus, it is possible to get an approximate view of the global state of the resources in the distributed system. This pattern requires that the kinds of resources to be monitored be defined *a priori*; if there is a need for adding new types of resources to be monitored at runtime, then one should use the **ADAPTIVE MONITOR (2)** instead. This pattern describes how to collect information about "**When**" to adapt, which will be used by the **EVENT DETECTOR (3)** for triggering the adaptations.

Related Patterns:

- The Publisher-subscriber pattern [Buschmann:1996] describes a mechanism for objects in a distributed system to declare interest in receiving information about a certain topic and for publishing information to be sent to the interested objects.

Known Uses:

- Grid computing systems instantiate this pattern to monitor the geographically distributed machines of the Grid. The Globus toolkit [Foster:1997], for example, uses the LDAP protocol for communicating information about resource availability and status of Grid nodes; a federation of LDAP servers plays the roles of the monitoring servers of this pattern. The InteGrade Grid middleware [Goldchleger:2003] also instantiates the pattern but uses CORBA for communication and a new service, called Global Resource Manager, as the monitoring server.
- The 2K operating system [Kon:2000, Kon:2005] instantiates this pattern to maintain an approximate view of the state of machines in the distributed system and uses this view as a hint for remote execution of user applications.
- The Framework for Adaptive Distributed Systems [Silva:2003] developed by Silva in his PhD work provides a generic object-oriented framework for instantiating this pattern based on CORBA distributed objects. Communication is performed with the CORBA event service, which is an instantiation of the Publisher-Subscriber pattern [Buschmann:1996].

Variant:

For some adaptive distributed applications there is no need for a centralized view of the state of distributed resources (e.g., a video client retrieving a movie from a video server). Instead, the application is only concerned with the state of resources located in the path between its components. In such a case, there is no need for a Monitoring Server. Each software component responsible for monitoring a resource should implement an interface through which the state of the resource can be queried. It should also implement a notification service through which applications can register interest in being notified about changes on the state of the monitored resource.

Implementation:

To implement this pattern, a system developer must implement and deploy Local Resource Monitors for each of the resources to be monitored. When implementing monitors for resources such as CPU and memory, almost always it is necessary to deal with the specificities of each operating system as there are no widespred standards for getting this kind of information. In Linux systems, for example, it is common to use the `/proc` pseudo-filesystem to get information about resource usage such as CPU and memory. In Windows systems, it is common to rely on Win32 API functions such as `GlobalMemoryStatus` to obtain memory information and `RegQueryValueEx` to get CPU consumption information from the Windows registry.

Local Resource Monitors must send data updates to the Monitoring Server periodically. This one-way communication can be implemented in various ways: from rudimentary sockets (using UDP, TCP or IP-Multicast channels) to higher level middleware mechanisms such as Java RMI, CORBA IIOP, or SOAP.

2. ADAPTIVE MONITOR

Motivation:

In very dynamic systems, we may not know *a priori* which objects and resources should be monitored. As new applications are installed in a system, we may need to monitor information that was previously irrelevant or not available.

Problem:

How to monitor system objects and resources that are not known at the design phase of the monitoring system?

Forces:

- Limiting the types of objects and resources that are monitored makes it easier to develop adaptive strategies because there is less information to be managed and analyzed. However, it is hard to predict ahead of time which objects and resources are available or needed as a system or demands on the system evolves.
- Systems, services, and resources change often as the requirements evolve. The monitoring system must cope with such changes by starting to monitor new things and stopping the monitoring of things that are no longer relevant.
- A distributed system is composed of several distributed resources, such as workstations, network links, and application servers. The usage of a resource can be expressed by different properties that adaptive applications might be interested in monitoring. For a network link, for instance, properties could be available bandwidth, current latency, and number of collisions. Therefore, a flexible way to describe distributed resources and their monitoring properties is needed.

Solution:

Define an Object Monitor responsible for gathering the state information of a single resource property and allow dynamic loading and unloading of these monitors in the various nodes of the distributed system. Each resource property could be defined with three attributes: resource name, property name and value type (e.g., <“Network Link”, “Available Bandwidth”, “Mbps”> <“CPU”, “CPU load”, “percentage”>).

Define a standard Object Monitor interface regardless of the property to be monitored and define a standard protocol and message format to be supported by the Monitoring Server. Each Object Monitor registers itself with the Monitoring Server as part of its instantiation process. Object Monitors send a message to the Monitoring Server whenever there is a significant change on the state of the resource properties they monitor, as illustrated in Figure 4.

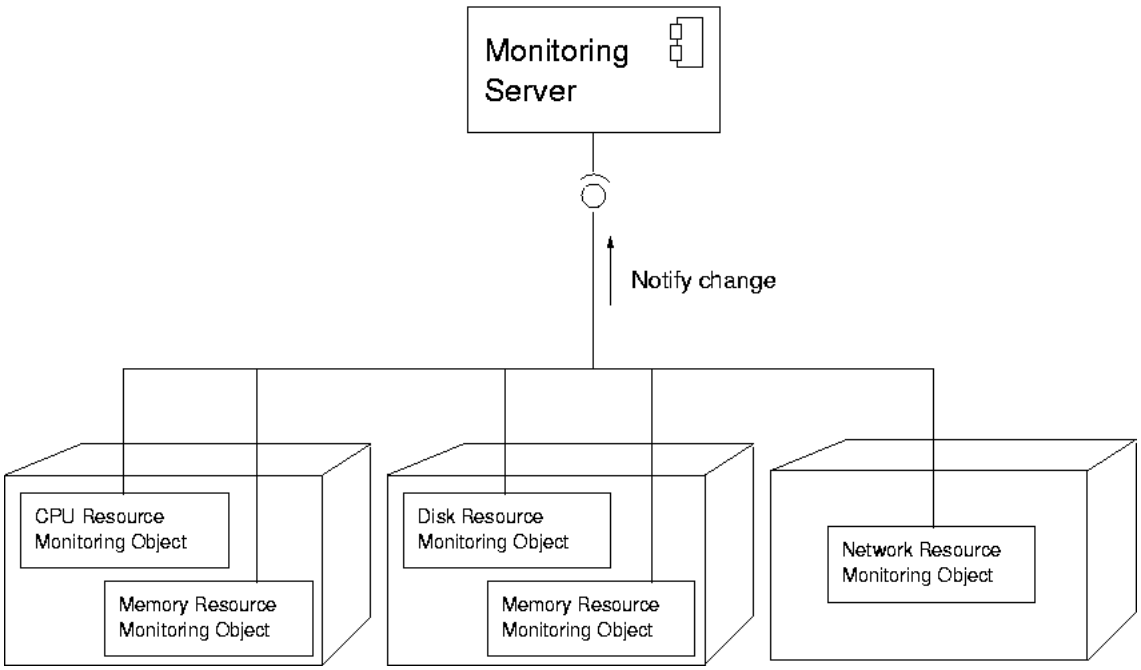


Figure 4 – Monitoring different kinds of resources

Example:

Consider the example described in the **DISTRIBUTED MONITOR (1)** pattern, where the CPU and memory of distributed machines are monitored to provide load balancing. When a machine becomes congested, with high CPU or memory usage, the load is balanced by migrating tasks to a machine with lower load. If new applications composed of several collaborative tasks must now be executed, a new resource property (available bandwidth) should also be monitored to avoid allocating collaborative tasks to different machines connected by low bandwidth links. Therefore, a new Object Monitor must be implemented (and dynamically loaded into the system) to monitor the available bandwidth.

As another example, consider a Web application for a bookstore. The load balancing among application servers can be based on CPU load. If, during the application execution, the administrator realizes that the network (and not the CPU) has become the bottleneck, an Object Monitor can be dynamically loaded into the application servers to redistribute the load based on the number of network connections rather than using just the CPU load as the scheduling parameter.

Consequences:

+ New or different resource properties can be monitored without affecting the code of the monitoring infrastructure
+ The monitoring service can start monitoring new or different resource properties without interrupting the service
– New resource properties should apply to the standard Object Monitor interface, which can limit the expressiveness of the resource property to be monitored

Implementation:

To implement such monitoring functionality, define a Resource Monitoring Object (RMO) that monitors a specific resource property. Each RMO monitors a single resource property that can correspond to physical resources such as memory, CPU, disk, and network links, but it can also monitor software parameters, such as the number of open threads in a server object.

Every resource property has a set of associated operation ranges, which are defined by the application developer. For example, one could use the following operation ranges for monitoring percentage of processor utilization: [0%, 10%), [10%, 25%), [25%, 50%), [50%, 75%), and [75%, 100%].

In all hosts containing resources that must be monitored, instantiate a Resource Monitoring Object for each resource property to be monitored. The RMO periodically verifies the current operation range of the resource property and only notifies registered components about changes on the operation range, limiting the number of monitoring messages in the distributed system. Figure 5 shows the RMO interface using CORBA IDL.

```
interface Rmo {
  MonitoredEntities::Parameter  parameter ();
  MonitoredEntities::Entity     me ();
  unsigned long    frequency ();
  unsigned long    current_range ();

  void  suspend ();
  void  resume ();
  void  change_frequency (in unsigned long new_frequency);
  oneway void  shutdown ();
};
```

Figure 5 - Resource Monitoring Object interface

The `parameter()` method returns a reference to the resource property being monitored while `me()` returns a reference to the monitored entity. `current_range()` returns the current operation range of the monitored property, allowing the developer to use, optionally, a pull approach in complement to the push mechanism described above. As an example, consider the operation ranges described above for monitoring percentage of processor utilization. The method `current_range()` would return 2 if a monitored processor usage is in the range [10%, 25%). The RMO interface also allows temporary suspension of the monitoring process (`suspend()`) as well as its resumption (`resume()`). `change_frequency()` alters the frequency used for verifying the operation range and `shutdown()` stops the monitoring process.

Figure 6 presents the class diagram for a Resource Monitoring Object responsible for monitoring the CPU usage on a host. The design is extensible, allowing the developer to construct easily new RMOs for monitoring other resource properties. To do so, the developer has to rewrite two classes, fully reusing the other five ones.

The `RMOImpl` class implements the Resource Monitoring Object interface illustrated in Figure 5. `CpuMonitor` and `RmoCpuImpl` are specific for the CPU usage property. The `CpuMonitor` class contains the code that actually verifies the

CPU usage with a given frequency encapsulated by the `Frequency` object. The user can change the frequency value through the `change_frequency()` method of the `RMOImpl` object. This method calls a `set()` method of the `Frequency` object. The user can also suspend or resume the monitoring by calling the `suspend()` and `resume()` methods of the `RMOImpl` object. These methods call the `SuspendMonitor` object that implements a monitor used by the `CpuMonitor` thread to verify if it must continue the CPU monitoring at the end of every monitoring interaction. A `CurrentRange` object encapsulates the value of the latest CPU usage calculated. The user can check the current CPU operation range by calling the `RMOImpl current_range()` method. The `Notifier` thread is responsible for sending a message to the Monitoring Server whenever there is a change on the CPU usage operation range.

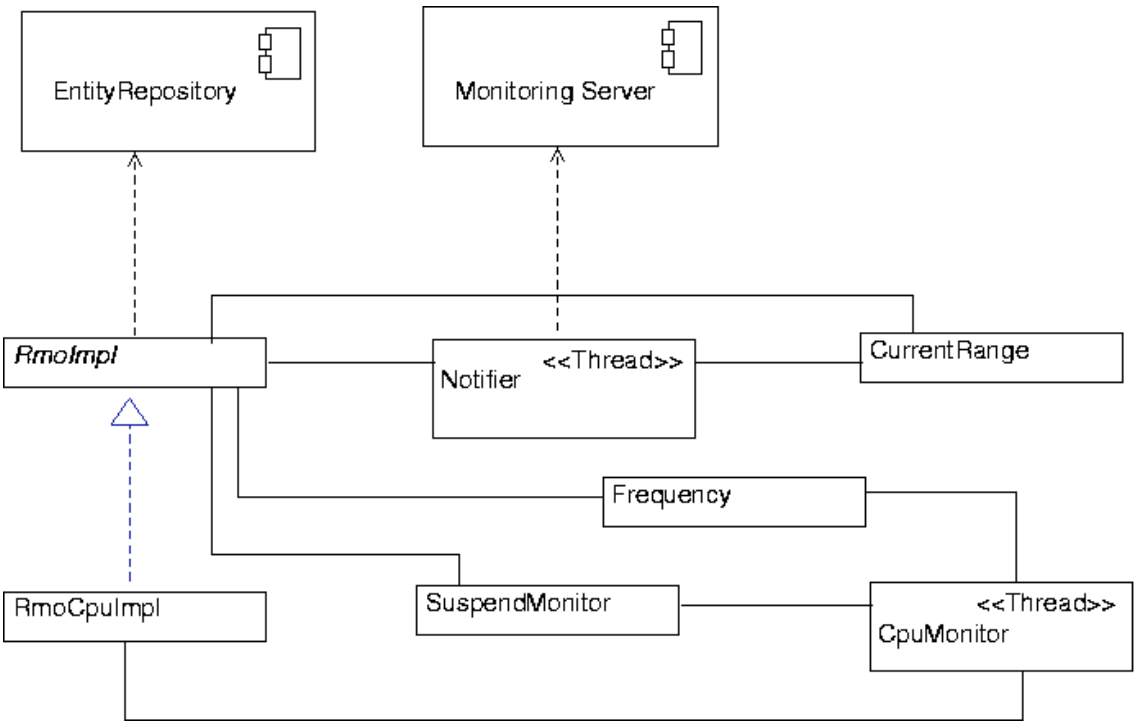


Figure 6 - Resource Monitoring Object monitoring CPU usage on a host

Resulting Context:

By applying this pattern, a collection of machines can be monitored with low overheads, enabling the construction of an approximate view of the global state of distributed resources. With the **ADAPTIVE MONITOR (2)**, the set of resources and the type of resources that are monitored can be reconfigured at runtime, enabling the monitoring of resources that were not anticipated at design time. This solution provides a large degree of flexibility. The data provided by the monitor will be processed by the **EVENT DETECTOR (3)**, which will trigger the adaptation actions.

Related Patterns:

- The Component Configurator pattern [Schmidt:2000] describes a mechanism for dynamically loading and configuring components into a running execution environment. This pattern can be used to load new monitoring objects dynamically.
- The TypeSquare pattern commonly used in Adaptive Object-Models (AOM) can be used for implementing the resource properties that can monitored [Yoder:2001; Yoder:2002]. The resource properties can be stored in a XML file that can be read at run-time in order to dynamically build the objects responsible for monitoring new defined resource properties without the necessity of recompiling and restarting the LocalResourceManager. AOM describes how to read the metadata file and dynamically build these objects using the Interpreter and Builder patterns [Gamma:1994].
- The Properties Pattern can be used for implementing different types of resource properties that are monitored [Foote:1998; Yoder:2001; Yoder:2002].

Known Uses:

- The Framework for Adaptive Distributed Systems [Silva:2003] allows specifying which resources will be monitored and how they will be monitored dynamically. This is achieved by dynamically loading new monitoring objects into the system runtime.
- The QuO Quality Objects Framework [Zinky:1997, Vanegas:1998, BBN:2002] provides a powerful CORBA-based framework for building quality of service aware, distributed applications. It instantiates this pattern using "system condition objects" as adaptable monitors.
- A mechanism for providing network environmental information in mobile wireless networks [Sudame:1997].
- An environment to support dynamic adaptation of distributed applications using the LuaORB system [Moura:2002].

3. EVENT DETECTOR

Motivation:

By analyzing the data provided by a distributed monitoring system, it is possible to identify relevant changes on resource availability that would impact application performance. By sending notifications to interested parties, it is possible to allow adaptive applications to reconfigure themselves to improve their performance in face of environmental changes.

Problem:

How to detect and notify applications about changes in the environment?

Forces:

- Tightly coupling the code responsible for detecting environmental changes with the application code adds unnecessary complexity, making the application code harder to implement, debug, and maintain. It also does not allow sharing the code with other environment-aware applications executing on the distributed system.
- On the other hand, each adaptive application can have specific needs concerning which environmental changes are relevant for dynamically adapting it.
- The notification of some environmental changes should be treated differently from others, leading to the need to apply different notification policies in different cases.

Solution:

An adaptive application must be notified of relevant changes in resource availability. These notifications can be implemented as asynchronous events. Expand the Monitoring Server interface (from the **DISTRIBUTED MONITOR (1)**) to allow the definition of event evaluators through conditional Boolean expressions. The Boolean expression indicates changes on resource property state. For instance, a "heavy use" event can be triggered when the percentage of the CPU usage on a host becomes greater than 80%.

Some applications need to correlate multiple events, such as the percentage of CPU usage and the amount of main memory available on a host. If this is the case, the definition of event evaluators must support composite events by allowing the Boolean expression to be composed of several resource properties.

Define an Event Channel to bind the event producer (Extended Monitoring Server) and event consumers (adaptive applications). The Event Channel implements the event delivery policy. You can organize your system with more than one Event Channel, each one responsible for notifying related events. For instance, a network channel may be associated with all network related events while a hardware channel may be associated with all events related to changes on hardware components. Figure 7 shows an abstract diagram of this architecture.

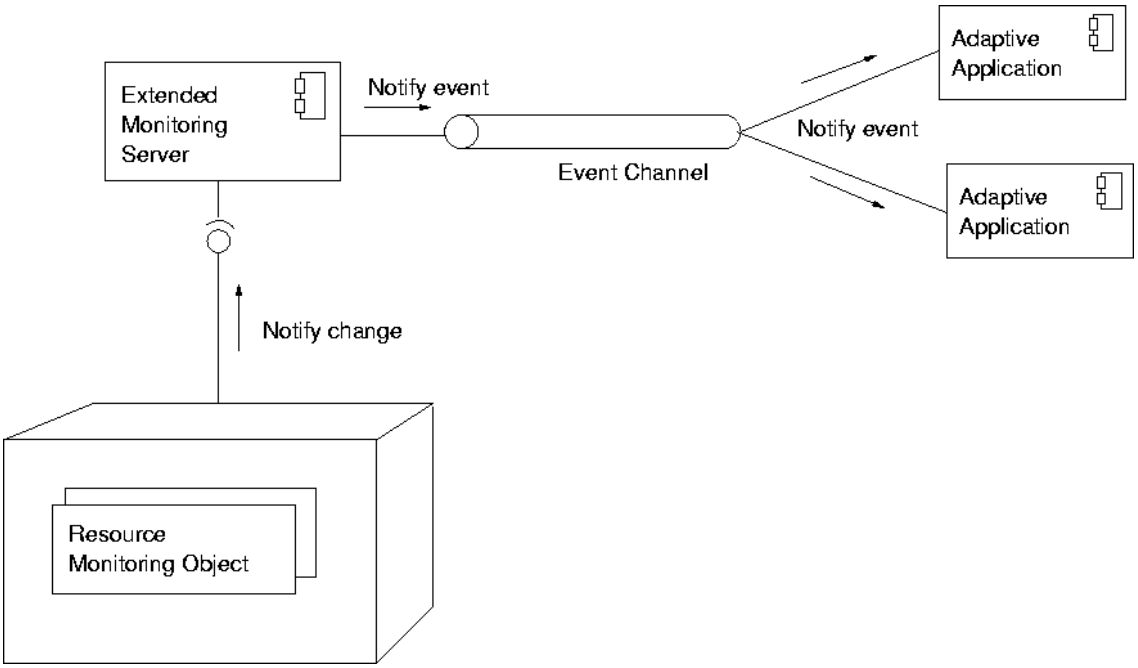


Figure 7 – Event Detector structure

Figure 8 illustrates the interactions between the pattern components. The adaptive application registers itself with the Event Channel, passing the list of environmental changes (events) in which it is interested. The resource monitoring objects (RMOs) continuously monitor the distributed system resources. Each RMO monitors a specific system parameter, notifying the Monitoring Server whenever a significant change is detected. The Monitoring Server evaluates all Boolean expressions containing the notified parameter and notifies the Event Channel whenever a Boolean expression is evaluated to true, meaning that an event has been detected. The Event Channel then notifies all adaptive applications that registered interest in the event that was triggered.

Example:

Consider a distributed system whose goal is to provide load balancing by migrating tasks from one machine to the other by looking at machines with congested CPUs and high memory usage. In each machine, instantiate two Object Monitors for monitoring the percentage of CPU load and the amount of memory available. Through the Monitoring Server interface, define a new Boolean expression that triggers an event every time a machine becomes congested, such as: `CPU_load > 80%` and `memory_available < 20MB`. Define an event channel used to notify the occurrence of events. The event channel abstraction is provided by some distributed object middleware services, such as the CORBA event service [CORBA:2002]. The channel uses a push approach, where the Monitoring Server registers itself as an event producer and the components of adaptive applications responsible for the dynamic reconfigurations register themselves as consumers.

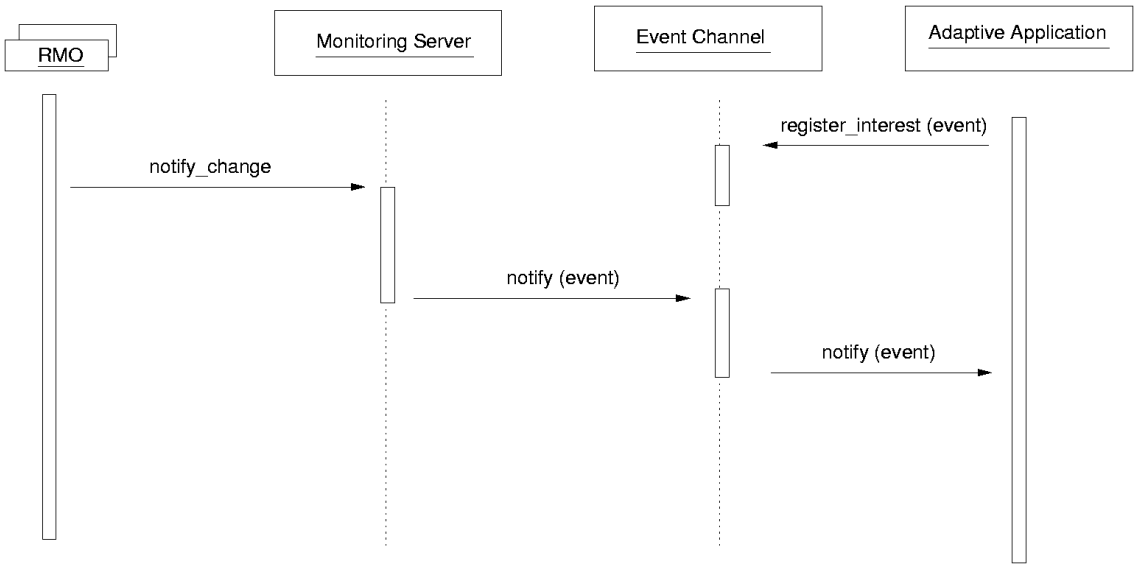


Figure 8 - Event detection and notification

Consequences:

+ Applications can share event evaluator definitions but also define new ones.
+ Allows flexible, event-specific delivery policies.
– Separating the code responsible for detecting environmental changes from application code adds complexity and can also lead to communication delays if they are deployed on separate machines.

Implementation:

Define a Monitoring Server responsible for collecting notifications of changes on the operation range of resource monitoring properties being monitored by Resource Monitoring Objects (from the **ADAPTIVE MONITOR (2)**). The Monitoring Server must allow the definition of events that are triggered based on a Boolean expression. Figure 9 shows a CORBA IDL interface for defining such resource events.

```
interface Event {

  string  eid();
  string  description();
};

interface ResourceEvent : Event {
  string                    expression ();
  unsigned long             duration_time ();
  MonitoredEntities::EntityType metype();
};
```

Figure 9 - IDL interface for an Event

`duration_time` specifies the amount of time that the Boolean expression must hold true to trigger an event notification. This avoids the notification of false events, based on temporary situations such as a short peak on CPU usage that occurs when a heavy application is started.

Since in a distributed system it is not guaranteed that messages are delivered in the same order that they were generated, each message from a RMO must include a timestamp. The Monitoring Server must maintain the last timestamp received from each RMO, discarding older messages without processing them.

Figure 10 shows the Monitoring Server interface. The `register()` method allows registering an event type and starts its detection. The interface allows the suspension and restart of the detection process though the `suspend()` and `resume()` methods, respectively. The `unregister()` method stops the detection of a given event type. An RMO notifies changes on the operation range of the resource being monitored through the `change_parameter()` method. To do so, it has to be previously registered through the `rmo_register()` method. If the execution of an RMO is suspended, resumed, or stopped the Monitoring Server must be informed through the `rmo_suspend()`, `rmo_resume()` and `rmo_unregister()` methods, respectively. The reason to do so, is that if an RMO is suspended or stopped, the Monitoring Server should not evaluate the event types whose Boolean expression contains the resource property that is no longer being monitored.

```
interface MonitoringServer{
      void register    (in ResourceEvent re);
      void unregister  (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
      void suspend     (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);
      void resume      (in string eid) raises(NoSuchEvent, EventNotBeeingEvaluated);

      void change_parameter (in string meid, in string pid,
                             in unsigned long new_range);

      void rmo_register   (in string strRmo, in string meid, in string pid);
      void rmo_unregister (in string meid, in string pid);
      void rmo_suspend    (in string meid, in string pid);
      void rmo_resume     (in string meid, in string pid);
      range_list list_me_parameter_range(in string meid);
    };
```

Figure 10 - MonitoringServer interface

The MonitoringServer interface uses the following terminology: a monitored distributed resource is called a monitored entity. An object representing every monitored entity and all its monitored parameters must be previously created through an Entity Repository. The Entity Repository will create a globally unique identifier for monitored entities and parameters. In the `MonitoringServer` interface, `eid` stands for event identification, `meid` stands for monitored entity identification and `pid` for parameter identification.

Figure 11 shows the class diagram of the Monitoring Server implementation. The `MonitoringServerImpl` class acts as a mediator. It receives information about resource utilization from remote `RMO`s and maintains a local list of entities being

monitored (instances of the `MonitoredEntity` class, which contains a description of the entity and its current value). `MonitoringServerImpl` also keeps a list of all active events whose descriptions are instances of `EventDescription`. An instance of the `Calculator` class performs the evaluation of the Boolean expressions defined in the event descriptions. Each time a Boolean expression is evaluated to true, a corresponding `SatisfiedEvent` is constructed. Once every second the `EventNotifier` checks for events whose expression holds true for the duration specified in the event definition. It then uses an Event Service (e.g., the CORBA one) to send notifications on event channels to which adaptive applications can listen to receive the notification of event occurrences.

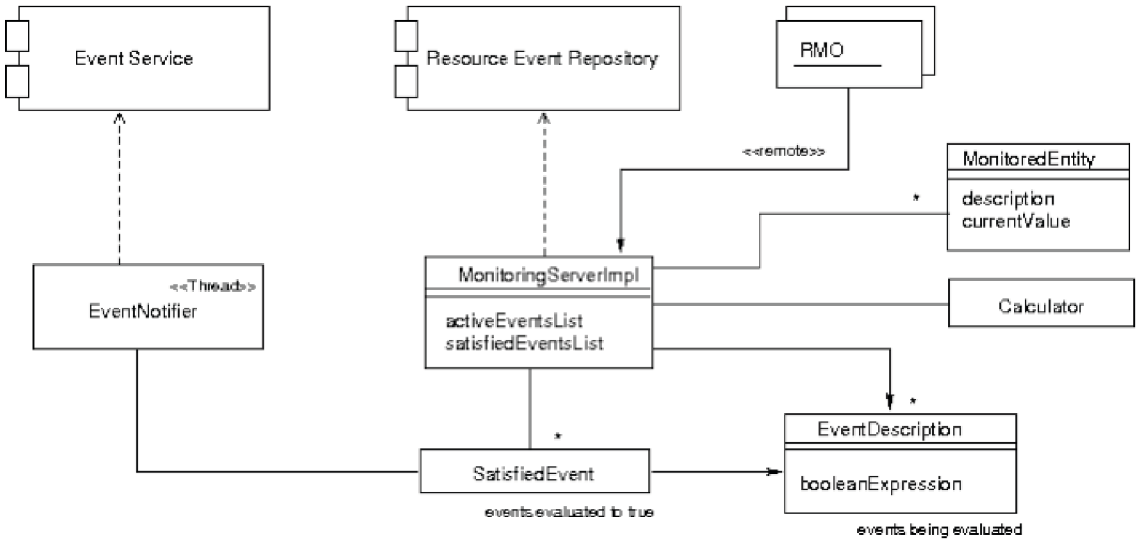


Figure 11 - Structure of the Monitoring Server implementation including event detection

Resulting Context:

By applying this pattern, it is possible to detect the occurrence of events based on the state of resources in a distributed system and to notify interested parties. In particular, the **AUTOMATIC RECONFIGURATOR (4)** and the **ADAPTIVE RECONFIGURATOR (5)** patterns rely on this event notification for carrying out the dynamic reconfiguration to adapt the applications.

Related Patterns:

- The instantiation of the Publisher-Subscriber pattern [Buschmann:1996] depends on a mechanism for matching subscriptions descriptions and event descriptions. This mechanism can be implemented by using the **EVENT DETECTOR (3)** described here.

- The Observer design pattern [Gamma:1994] describes an even simpler notification mechanism that can be used in some cases.
- The TypeSquare pattern described in Adaptive Object-Models (AOM) can be used for implementing the set of events that can change at runtime [Yoder:2001; Yoder:2002]. Event definitions can be stored in a XML file that can be read at run-time in order to dynamically build the objects responsible for detecting new defined events without the necessity of recompiling and restarting the MonitoringServer. AOM describes how to read the metadata file and dynamically build these objects using the Interpreter and Build patterns.
- The Interpreter design pattern [Gamma:1994] define a representation for a given language grammar along with an interpreter that uses the representation to interpret sentences in the language. It can be used for implementing the Boolean expression evaluator.

Known Uses:

- The Framework for Adaptive Distributed Systems [Silva:2003] includes an engine for event detection that instantiates this pattern faithfully.
- The QuO Quality Objects Framework [Zinky:1997] uses "delegates" and "contracts" to instantiate this pattern. Delegates act as proxies [Gamma:1994, Buschmann:1996] that intercept remote method calls; during interception, a delegate evaluates a contract to detect possible contract violations, which can be seen as a form of event detection.
- Moreto and Endler [Moreto:2001] describe a general purpose Event Processing Service (EPS), which can be used to detect primitive and composite events. Composite events are defined through an event expression based on primitive event types combined by a set of operators, similarly to the **EVENT DETECTOR (3)**.
- Welling and Badrinath [Welling:1997] describes an architecture for exporting environment awareness to mobile computing applications. In their architecture, a change in the environment is modeled as an asynchronous event that includes information related to the change. The architecture also allows alternate event delivery policies by isolating the event delivery functionality within a channel, as done in the **EVENT DETECTOR (3)**.

4. AUTOMATIC RECONFIGURATOR

Motivation:

A distributed system has many resources whose availability and load vary intensely. To cope with these variations, an adaptive application must be able to reconfigure itself as relevant changes on resource availability occur. Changes on resource availability could be notified through an event distribution mechanism. For each event, the set of adaptive actions that must be performed may vary.

Problem:

Given event notifications indicating changes on resource availability, how can a system apply dynamic reconfiguration actions automatically without the need for any human interference?

Forces:

- Coupling the application functional code with the code responsible for dynamic adaptation increases code complexity, making it harder to implement, debug, and maintain.
- The application developer should concentrate on the application core functionality, considering the adaptation issues as a separate aspect.
- Limiting which adaptation mechanisms can be applied (e.g., adjusting application parameters, switching between algorithms and relocation or replication of application components) restricts the solution applicability.

Solution:

Using a reflective model [Maes:1987], organize the application in two levels: a meta-level composed of objects responsible for receiving notifications of events describing environmental changes and for applying the reconfiguration actions and a base-level, that deals with regular application functionality.

For each event type indicating environmental changes that requires adaptation, describe the reconfiguration action(s) that your application must apply and code it into objects (called Handlers) that will be part of the meta-level. Each Handler object must implement a `run()` method, responsible for applying the reconfiguration actions when called. As illustrated in Figure 12, the Event Handler registers itself with the Event Channel (described in the **EVENT DETECTOR (3)**). Through the Event Channel, it receives notifications of environmental changes and reacts to them by applying the reconfiguration actions coded in its `run()` method.

Adaptive actions can be based on several mechanisms, such as:

a) Adjusting parameters of base-level objects (e.g., changing the presentation rate of video frames as network bandwidth varies);

b) Switching between algorithms used by base-level objects (e.g., changing a compression algorithm as CPU usage and network bandwidth varies);

c) Relocating or replicating application components to other network nodes.

In the application meta-level, instantiate an active object that registers itself as a consumer of the network event channel responsible for notifying environmental changes. Upon the receipt of an event, it calls the `run()` method of the corresponding Handler object.

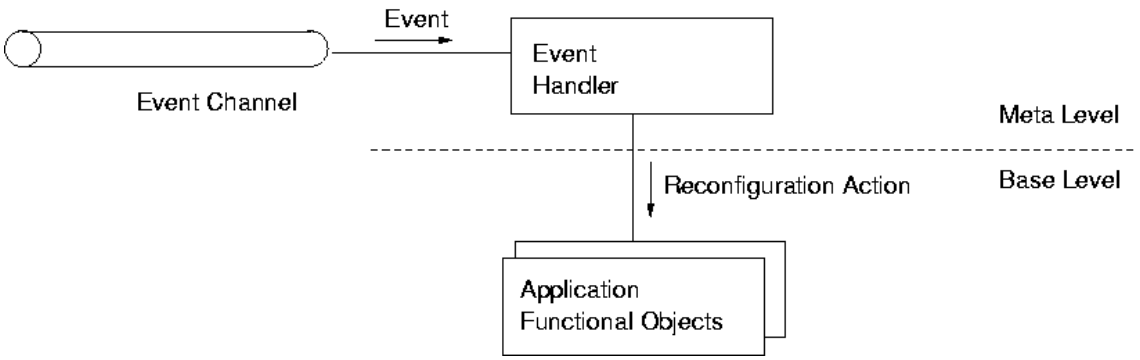


Figure 12 – Automatic Reconfigurator structure

Example:

Consider again the load balancing problem based on machine load. In the application meta-level, instantiate an object that registers itself as a consumer of the event channel responsible for notifying environmental changes. The adaptive application must react to a notification of an overloaded machine by migrating a subset of the tasks executing at the congested location to a machine with better CPU and memory availability (if one is available). The migration should be triggered by the `run()` method of the Handler object. This reconfiguration can be done with the assistance of libraries that support the reconfiguration of distributed applications [KonPhD:2000].

Consequences:

+ Leads to a clear separation of concerns between the application functional code and the adaptation code. As a consequence, the resulting application becomes easier to design, implement, and maintain.

_ The actions applied in reaction to an environmental change are hard-coded into the application and cannot be dynamically changed.

_ Reflection adds complexity to the system which can make understanding and maintaining the system harder.

Implementation:

To implement this pattern, first it is necessary to make the event handlers capable of receiving event notifications by using a mechanism compatible with the notification mechanism chosen in the implementation of the **EVENT DETECTOR (3)**.

Then, it is required to implement handlers capable of changing the internal behaviour or structure of the application. Thus, normally the handler programmer must have a very good knowledge of the application implementation. One way to mitigate this requirement is to provide an interface in the application that programmers can use to set some parameters that determine how the application works. In this case, the application programmer would specify how the application could be configured (by specifiying which parameters can be set) and the event handler programmer would simply write the code that sets the proper values for the parameters.

Resulting Context:

By applying this pattern, the system is able to respond to changes in the environment by reconfiguring the applications, allowing for the implementation of self-adaptive applications. The set of reconfiguration actions to be taken are hard-coded and cannot be modified without recompiling the application. If the ability to dynamically redefiny the reconfiguration actions is desirable, then the **ADAPTIVE RECONFIGURATOR (5)** pattern should be used instead.

Related Patterns:

- The Reflection architectural pattern [Buschmann:1996] describes how to separate components of an adaptable system in a meta-level and a base-level. Base-level components deal with functional aspects of the system while meta-level components deal with non-functional aspects such as dynamic reconfiguration.
- Foote and Yoder [Foote:1995] describe some common reflective patterns that should be considered when building systems that need to be able to dynamically adapt at runtime.

Known Uses:

- The Video Datagram Protocol used by the Vosaic system [Chen:1996] uses a hard-coded adaptation algorithm that changes parameters of a video streaming session based on the monitored rate of dropped network packets.
- Chang and Karamcheti [Chang:2000] describe a framework for automatic configuration and run-time adaptation of distributed applications that hard-code alternative execution paths (algorithms) that are dynamically selected by guard expressions of control parameters.
- Noble and Satyanarayanan [Noble:1999] describes Odyssey, a platform for mobile data access. The adaptive application is divided in client and server components and the hard-coded adaptation mechanism allows to choose between different versions of the data being retrieved, so as to be compatible with the environment resource availability.

5. ADAPTIVE RECONFIGURATOR

Motivation:

An adaptive application must change its behavior in reaction to changes in its execution environment. The application can consider several different events that signal environmental changes; for each event, the adaptive actions that must be performed can vary. More flexibility can be achieved if the developer is allowed to specify collections of adaptive actions (adaptation policies) for each event, switching from one to another at run time depending on the application context. The adaptation mechanism can also evolve or be debugged without restarting the application if it allows dynamic loading and unloading of adaptation policies.

Problem:

How to vary at run time the collection of adaptive actions?

Forces:

- On the one hand, statically configuring the adaptation policies into the application code requires stopping, recompiling, and restarting the application whenever new code for an adaptation policy or changes in an old one are developed. These activities are infeasible for applications with high availability requirements. Dynamic loading and unloading of adaptation policies not only resolves this problem but also minimizes resource consumption, since only policies that are in use need to be loaded into the application code. This is particularly important if the computer on which the application is running has memory and processing limitations.
- On the other hand, the dynamic loading and unloading of adaptation policies implies application overhead.

Solution:

Decouple the Event Handler interface from its implementation by defining a uniform interface for each Event Handler that applies adaptive actions for a given environmental event. Develop one or more concrete Event Handlers that implement this interface. Each concrete Event Handler implements an adaptation policy. As illustrated in Figure 13, define for each application component a corresponding `Component Configurator` [KonPhD:2000]. The `Component Configurator` keeps track of the dynamic dependencies between the component and other system or application components and is also responsible for (1) disseminating events across inter-dependent components, whenever they affect several correlated components and (2) carrying out component-specific reconfiguration actions by calling component methods directly. Through this mechanism, the Event Handler can propagate the adaptive actions required to adapt the application to the new environmental state.

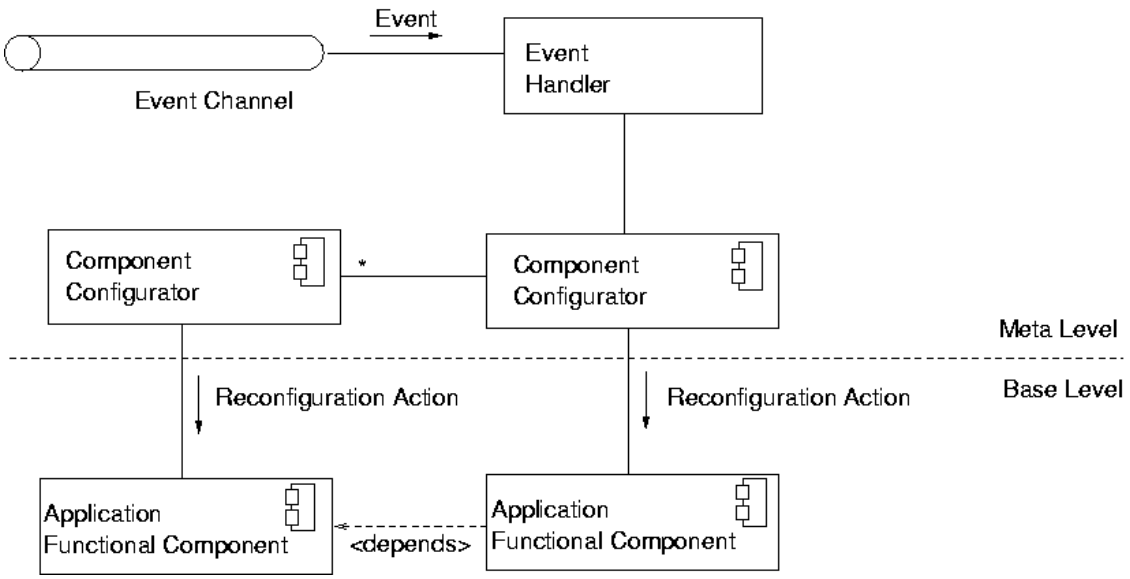


Figure 13 – Adaptive Reconfigurator structure

Example:

It is possible to design an application to be adaptable in ways that can be fully specified at design time, but it is difficult, if not impossible, to anticipate all the ways in which it may be required to adapt some applications. For instance, in mobile computing environments, the characteristics of the network connections can range from an inexpensive, very high bandwidth with low latency connection such as high-speed LAN, to a very expensive, low bandwidth with high latency connection such as GSM or infrared. Even the network address of the machine can change. Mobile applications should also be able to handle periods of disconnection. The application and data characteristics, and the user's context requirements and limitations may all change dynamically. Any of these contextual conditions can change without warning and to values unknown and unforseen by the application designer. Thus, it might be necessary to load new adaptation policies at runtime.

Consequences:

+ Several adaptation policies can be defined and reconfigured at run time for each environmental event.
+ The adaptation policies can be loaded and unloaded dynamically, allowing the adaptation mechanism to evolve or be debugged without restarting the application. This also minimizes resource consumption.
_ The dynamic loading and unloading of adaptation policies generates overhead to the adaptation mechanism.
_ The level of complexity increases making applications more difficult to develop and maintain.

Implementation:

Organize application components in two layers: (1) a metalevel layer, responsible for receiving event notifications describing changes on distributed resource usage and also

applying reconfiguration actions to adapt the application to the new environmental state; and (2) a base level, that provides the application functionality.

Define, for each application component, a corresponding `Component Configurator` object. As described in the Dynamic Dependence Manager pattern [Domingues, 2005], the `Component Configurator` keeps track of the dynamic dependencies between the component and other system or application components and helps to maintain runtime consistency in the presence of reconfigurations. `Component Configurators` are also responsible for disseminating events across inter-dependent components. Examples of common events are the failure or migration of a component, internal reconfiguration, or replacement of the component implementation. The rationale is that those events affect all the dependent components. This communication mechanism coordinates reconfiguration actions among the application components. The `Component Configurator` contains the code to deal with these configuration-related events. This approach provides a clear separation of concerns between the application functional code and the code that deals with the application reconfiguration.

As illustrated in Figure 14, create an `EventReceiver` that registers itself with the event channels (described in the **EVENT DETECTOR (3)**). The `EventReceiver` will be notified of events that indicate relevant environmental changes. Depending on the type of the event, it executes the appropriate actions required to adapt the application to the new environment state, using the `Component Configurators` to coordinate reconfiguration actions among the, possibly distributed, application components. Organize the classes that handle each environment event as a set of Event Handler strategies, using the Strategy design pattern. Figure 14 illustrates three strategies `(EventAHandler1, EventAHandler2, and EventAHandler3)` that can be triggered when an instance of `EventA` is notified.

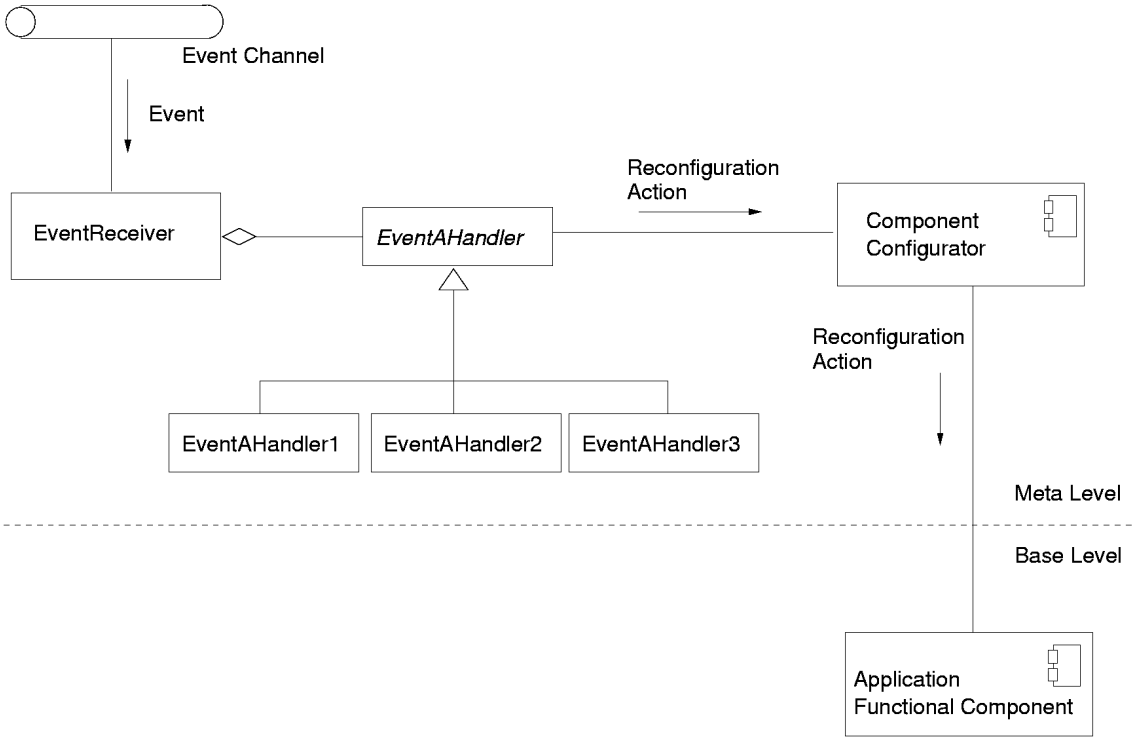


Figure 14 - Architecture for handling events and reconfiguring the application

Concrete Event Handlers should be packaged into a suitable unit of configuration that can be dynamically linked to the application, such as a dynamically loaded library (DLL) or a Java class file. The dynamic loading and unloading of Event Handlers can be controlled by specific configuration mechanisms such as the Component Configurator (forming a metametalevel, not illustrated in Figure 15). Other configuration operations, such as suspend and resume, can also be supplied. Suspending the execution of an Event Handler implies not executing the reconfiguration actions when the corresponding event is triggered. Resuming it, turns back the adaptive behavior of the application for the corresponding event.

Resulting Context:

By applying this pattern, the system is able to respond to changes in the environment by reconfiguring the applications, enabling the implementation of self-adaptive applications. In addition, the set of adaptation actions is also reconfigurable, allowing the application maintainer or operator to modify or add new reconfiguration strategies at runtime. This permits the construction of highly flexible and reconfigurable applications that can evolve and change radically at runtime without the need for shutdown and restart.

Related Patterns:

- The Strategy design pattern [Gamma:1994] explains how to build a system such that the algorithms it uses can be changed dynamically. The **ADAPTIVE RECONFIGURATOR (5)** can be implemented by enhancing an implementation of the **AUTOMATIC RECONFIGURATOR (4)** with the Strategy pattern.
- The Component Configurator pattern [Schmidt:2000] (not to be confused with the Component Configurator object [KonPhD:2000] used in the implementation section of the pattern described here) describes a mechanism for dynamically loading and configuring components into a running execution environment. This pattern can be used to load new Event Handlers dynamically allowing new forms of reconfiguring a system.
- The TypeSquare pattern described in Adaptive Object-Models (AOM) can be used for implementing the set of events that an extended ComponentConfigurator can handle [Yoder:2001; Yoder:2002]. Event definitions can be stored in a XML file that can be read at run-time in order to dynamically build the objects responsible for handling new defined events without the necessity of recompiling and restarting the extended ComponentConfigurator. AOM describes how to read the metadata file and dynamically build these objects using the Interpreter and Build patterns.

Known Uses:

- The Framework for Adaptive Distributed Systems [Silva:2003] allows dynamic loading new Component Configurators and new Event Handlers at runtime.
- The dynamicTAO reflective ORB [Kon&Roman:2000] allows dynamic loading new ORB components at runtime, including components that take care of the reconfiguration process itself.

Variant:

If the adaptation mechanism must provide more than one adaptation policy for a given event but dynamic loading and unloading its code is unnecessary, the Strategy pattern can be applied instead of the Component Configurator. Decouple the Event Handler interface from its implementation and develop concrete Event Handlers that implement the uniform interface. Each concrete Event Handler implements an adaptation policy and corresponds to a Concrete Strategy using the Strategy pattern terminology. A Context object must be configured with the concrete Handler to be used when the corresponding event is triggered. If the adaptation mechanism must switch from one adaptation policy to another at run time, all concrete Event Handlers must be instantiated as part of the application initialization. If only one policy will be used in a single application execution, it is only necessary to instantiate the Event Handler that corresponds to the policy that will be applied.

Pattern Language Summary

Computing environments today require systems to adapt quickly to changes, which often includes reconfiguring or adapting to an evolving environment. This paper presented a pattern language for assisting with this requirement, specifically with the problem of building automatically configurable and adaptive distributed systems. The pattern language outlines an architecture for describing "**When**" should an adaptation be done (monitors), "**What**" adaptation should be performed (event detection) and "**How**" to adapt the system (reconfigurators).

The **DISTRIBUTED MONITOR (1)** provides a simpler solution for monitoring distributed resources while the **ADAPTIVE MONITOR (2)** describes an extension that supports dynamic reconfiguration of the monitor. The approach uses rules for triggering events for when adaptations should be performed. The **EVENT DETECTOR (3)**, uses the rules and notifies the mechanisms responsible for the dynamic reconfiguration of the system. The reconfigurators provide a mechanism for actually adapting the system safely according to the specified rules. The **AUTOMATIC RECONFIGURATOR (4)** describes a simpler solution while the **ADAPTIVE RECONFIGURATOR (5)** describes dynamic reconfiguration of the reconfiguration process, thus making the reconfigurator more adaptable and reconfigurable.

There are orthogonal issues that will need to be addressed while applying these patterns such as Security, Fault-Tolerance, and Real-Time, which are beyond the scope of this pattern language.

Acknowledgments

The authors would like to thank Eugene Wallingford, Paulo Borba, Linda Rising, Giuliano Mega, and Eduardo Fernandez for their valuable thoughts and suggestions that greatly contributed to this work. We would also like to thank the Software Architecture Group from the University of Illinois at Urbana-Champaign and the SugarLoafPLoP'2005 Araucaria group for their valuable feedback and comments.

References

[BBN:2002] BBN Technologies. *QuO ToolKit User's Guide*, release 3.0.10, April 2002. http://quo.bbn.com.

[Buschmann:1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[Chang:2000] Fangzhe Chang and Vijay Karamcheti. Automatic conguration and run-time adaptation of distributed applications. In *Ninth IEEE International Symposium on High Performance Distributed Computing*, pp. 11-20, Pittsburg, Pennsylvania, August 2000.

[Chen:1996] Zhigang Chen and See-Mong Tan and Roy H. Campbell and Yongcheng Li. Real-Time Video and Audio in the World Wide Web. In *World Wide Web Journal*. 1(1). 1996.

[CORBA:2002] OMG - Object Management Group. *The Common Object Request Broker: Architecture and Specication*, November 2002. version 3.0.1.

[Domingues:2005] Helves Domingues and Marco A. S. Netto. *The Dynamic Dependence Manager Pattern*. Technical Report RT-MAC-2005-07, Department of Computer Science, University of São Paulo. 2005.

[Foote:1995] Brian Foote and Joseph W. Yoder Evolution, Architecture, and Metamorphosis. In *Second Conference on Patterns Languages of Programs (PLoP '95)*. Monticello, Illinois, September 1995. Also *Pattern Languages of Program Design 2* edited by John M. Vlissides, James O. Coplien, and Norman L. Kerth. Addison-Wesley, 1996.

[Foote:1998] Brian Foote and Joseph Yoder. *Metadata and Active Object-Models Collected papers from the PLoP '98 and EuroPLoP '98 Conference*, Technical Report WUCS-98-25, Department of Computer Science, Washington University, September 1998.

[Foster:1997] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *International Journal of Supercomputer Applications*. 11(2), pp. 115-118. 1997.

[Gamma:1994] Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[Goldchleger:2003] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. In *Concurrency and Computation: Practice & Experience*. Vol. 16, pp. 449-459. March, 2004.

[Kircher:2004] Michael Kircher, Prashant Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*. John Wiley & Sons, 2004.

[Kon&Roman:2000] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Conguration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121-143, New York, April 2000. Springer-Verlag.

[Kon:2000] Fabio Kon, Roy Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *9th IEEE International Symposium on High Performance Distributed Computing*. Pittsburgh. August 1-4, 2000.

[KonPhD:2000] Fabio Kon. Automatic *Conguration of Component-Based Distributed Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2000.

[Kon:2005] Fabio Kon, Jeferson Roberto Marques, Tomonori Yamane, Roy H. Campbell, and M. Dennis Mickunas. Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems. In *Software: Practice and Experience*, 35(7), pp. 667-703, May 2005.

[Maes:1987] Maes P. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications Conference '87*, volume 22 of Sigplan Notices, pages 147-155. ACM, December 1987.

[Moreto:2001] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. In *IEEE Proceedings Software*, 2001. ISSN 1462-5970, 148(1).

[Moura:2002] Moura A, Ururahy C, Cerqueira R, and Rodriguez N. Dynamic support for distributed auto-adaptive applications. In *Proceedings of AOPDCS - Workshop on Aspect Oriented Programming for Distributed Computing Systems (held in conjunction with IEEE ICDCS 2002)*, pages 451-456, Vienna, Austria, July 2002.

[Noble:1999] B. D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in Odyssey. In *Mobile Networks and Applications*, 4(4):245-254, 1999. Kluwer.

[Schmidt:2000] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. John Wiley & Sonss, 2000.

[Silva:2003] Francisco J. S. Silva, Markus Endler, and Fabio Kon. Developing Adaptive Distributed Applications: a Framework Overview and Experimental Results. In *Proceedings of the International Symposium on Distributed Objects and Applications*. LNCS 2888, pp.1275-1291. Catania, Sicily, Italy, November, 2003.

[Sudame:1997] Sudame P and Badrinath B. *On providing support for protocol adaptation in mobile wireless networks*. Technical report, Department of Computer Science, Rutgers Universit, June 1997. http://www.cs.rutgers.edu/pub/technical-reports/dcstr-333.ps.Z.

[Vanegas:1998] Vanegas R, Zinky J, Loyall J, Karr D, Schantz R, and Bakken D. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England, September 1998.

[Welling:1997] Girish Welling and B. R. Badrinath. A framework for environment aware mobile applications. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*, May 1997.

[Yoder:2001] Joseph Yoder, Federico Balguer and Ralph Johnson. Architecture and Design of Adaptive Object-Models. In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*. ACM SIGPLAN Notices, December 2001.

[Yoder:2002] Joseph Yoder and Ralph Johnson. The Adaptive Object-Model Architectural Style. In *Proceedings of the Workshop IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02)* at the World Computer Congress in Montreal, August 2002. Software Architecture System Design, Development and Maintenance Edited by Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela; Kluwer Academic Publishers 2002.

[Zinky:1997] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural Support for Quality of Service for CORBA Objects. In *Theory and Practice of Object Systems*. April, 1997.

Padrões de Requisitos para Especificação de Casos de Uso em Sistemas de Informação

Gabriela T. de Souza[1, 2], Carlo Giovano S. Pires [2] e Arnaldo Dias Belchior[1]

[1]Universidade de Fortaleza
Av. Washington Soares, 1321 – Fortaleza – CE – Brasil

[2]Instituto Atlântico
Rua Chico Lemos, 946 – 60 822-780 – Fortaleza – CE – Brasil

belchior@unifor.br, {gabi,cgiovano}@atlantico.com.br

Abstract. *This work presents a set of requirement patterns for information systems. These patterns are based on the use case concept and present solutions for use cases specification problems, considering maintenance operations (insert, update and delete), transaction and query functionalities, which are a representative part of information systems scope.*

Resumo. *Este trabalho apresenta um conjunto de padrões de requisitos para sistemas de informação. Esses padrões são fundamentados no conceito de casos de uso e apresentam soluções para problemas de especificação de requisitos funcionais, considerando operações de manutenção (inclusão, alteração e exclusão), transação e consulta, que representam um volume significativo do escopo de sistemas de informação.*

1. Introdução

Este trabalho apresenta um conjunto de padrões de requisitos para sistemas de informação, que são fundamentados no conceito de casos de uso. Esses padrões abordam soluções para problemas de especificação de requisitos funcionais considerando questões de operações de manutenção, consulta, relatório e operações de transação. Isto representa um volume significativo do escopo de sistemas de informação.

O relacionamento entre os padrões apresentados pode ser visto na Figura 1. Nesta figura, os retângulos representam os padrões e as setas representam que os padrões que se encontram na origem da seta usam o padrão que se encontra no destino da seta.

Caso de uso é um conceito amplamente difundido e utilizado para a documentação e o desenvolvimento de requisitos [3] [4] [5] [6] [7]. Segundo o RUP [2], caso de uso é uma descrição de comportamento do sistema em termos de seqüências de ações. Um caso de uso deve produzir um resultado de valor observável para um ator. Ele contém todos os fluxos de eventos referentes à produção do "resultado de valor observável". Mais formalmente, um caso de uso define um conjunto de instâncias de

casos de uso ou cenários [2]. O CMMI indica que casos de uso podem ser usados na elicitação e análise de requisitos para estabelecer os cenários operacionais do sistema [1]. Ou seja, além de representar os requisitos, os casos de uso também descrevem uma solução em alto nível.

Este trabalho utiliza um formato de caso de uso definido pela Rational [2], que compreende seções como fluxos básicos e alternativos, subfluxos de execução, requisitos especiais e regras de negócio. Serão apresentados os seguintes padrões: (i) padrão **Caso de Uso CRUD**; (ii) padrão **Documentação de Atributos**; (iii) padrão **Caso de Uso Relatório**; (iv) padrão **Caso de Uso Transação** e (v) padrão **Caso de Uso Assistente**.

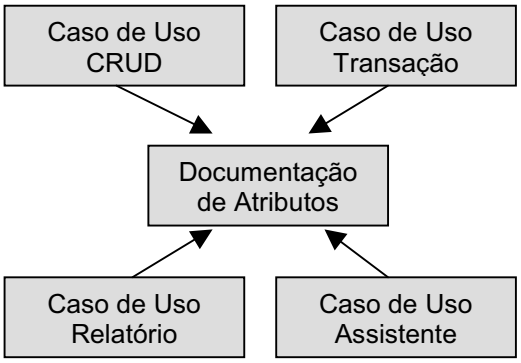


Figura 1: Relacionamento entre os padrões apresentados

2. Caso de Uso CRUD

2.1. Contexto

Este padrão é utilizado para a documentação dos requisitos de operações de manutenção em sistemas da informação, por meio do uso de modelos e especificações de casos de uso. Os requisitos de operações de manutenção são caracterizados por operações de Inclusão, Consulta, Alteração e Exclusão.

2.2. Problema

Como documentar os requisitos funcionais de inserção, atualização, exclusão e consulta de dados por meio de especificações de casos de uso?

2.3. Forças

- Todo caso de uso deve demonstrar um valor observável [2]. Em alguns casos, o usuário identifica o valor observável como a manutenção da entidade. Em outros casos, o valor observável está nas operações individuais de Inclusão, Consulta, Alteração e Exclusão.
- As operações de manutenção podem ocorrer tanto sobre entidades simples, com poucos atributos, como em entidades complexas, com vários atributos e relacionamentos.
- As operações de inclusão, alteração, remoção e consulta devem ser tratadas e seus requisitos documentados. Esses requisitos incluem validação de atributos e regras de negócio.

- Os atributos mantidos de cada entidade devem ser documentados.
- Os requisitos documentados devem ser de fácil entendimento para os usuários e para a equipe de desenvolvimento.
- Uma quantidade grande de casos de uso dificulta a gestão dos requisitos e pode indicar a existência de decomposição funcional.

2.4. Solução

Organizar o fluxo de eventos do caso de uso em cinco subfluxos (**Fluxo básico, Incluir, Alterar, Remover e Consultar**) como se segue:

- O **Fluxo básico** descreve a condição de início e desvia o fluxo para um dos subfluxos, de acordo com as operações disponíveis: Incluir, Alterar, Excluir e Consultar. Condições de início indicam os eventos que provocam a execução do caso de uso. Por exemplo, em que situação a entidade deve ser mantida, se existe alguma periodicidade requerida ou alguma questão de permissão de acesso.
- Cada subfluxo descreve o cenário operacional de uma das funcionalidades: Incluir, Alterar, Remover e Consultar.
- O subfluxo **Incluir** apresenta os atributos para a inclusão e descreve o comportamento da inclusão.
- O subfluxo **Alterar** apresenta os atributos atualizáveis, exibe seus valores e descreve o comportamento da atualização. Se os atributos atualizáveis forem os mesmos apresentados no subfluxo **Incluir,** pode-se referenciar este subfluxo.
- O subfluxo **Remover** descreve o comportamento da remoção e documenta as restrições da exclusão da entidade. Por exemplo, se alguma regra de negócio deve ser acionada ou se uma confirmação para a exclusão é exigida.
- O subfluxo **Consultar** documenta requisitos para localização da entidade, que atributos devem ser filtros para a consulta, quais são obrigatórios e quais atributos são exibidos no resultado.
- As validações de atributos e regras de negócio são documentadas em uma seção independente dos fluxos e subfluxos, ver o padrão **Documentação de Atributo**. A decisão sobre o momento no qual as validações e regras são executadas fará parte do projeto do caso de uso. No entanto, se esse momento já for identificado como um requisito claro da aplicação, a regra ou validação deve ser referenciada pelo subfluxo. As regras de negócio, tipicamente, representam requisitos de cálculos e tratamento de relacionamentos com outras entidades. As validações, tipicamente, documentam o tratamento para a obrigatoriedade de atributos e o tratamento de formato de atributos (datas, limites numéricos, entre outros).

2.4.1 Estrutura

Fluxo básico

1. O caso de uso inicia quando o <*nome do ator*> necessita fazer a manutenção (inclusão, alteração, exclusão ou consulta) de uma <*nome da entidade*>. <*descrever a condição de início do caso de uso*>
2. De acordo com o tipo de operação manutenção desejado pelo <*nome do ator*>, um dos subfluxos é executado:
    a. Se o <*nome do ator*> deseja incluir uma nova <*nome do ator*> , o subfluxo Incluir <*nome do ator*> é executado.
    b. Se o <*nome do ator*> deseja alterar informações de uma <*nome do ator*> já cadastrada, o subfluxo Alterar <*nome do ator*> é executado.
    c. Se o <*nome do ator*> deseja excluir uma <*nome do ator*> já cadastrada, o subfluxo Remover <*nome do ator*> é executado.
    d. Se o <*nome do ator*> deseja consultar informações sobre uma ou mais <*nome do ator*> cadastradas, o subfluxo Consultar <*nome do ator*> é executado.

Subfluxo Incluir <nome da entidade>

1. Este subfluxo inicia quando o <*nome do ator*> solicita incluir uma <*nome da entidade*>;
2. O sistema solicita ao <*nome do ator*> o preenchimento dos seguintes atributos:
    - <*lista de atributos*>.
3. O <*nome do ator*> preenche os atributos acima e confirma a inclusão;
4. O sistema realiza a inclusão dos dados informados pelo <*nome do ator*> no passo 3;
5. O sistema exibe uma mensagem informando que a inclusão da <*nome da entidade*> foi efetivada com sucesso;

Subfluxo Alterar <nome da entidade>

1. Este fluxo inicia quando o <*nome do ator*> solicita alterar uma <*nome da entidade*>;
2. O <*nome do ator*> seleciona um único <*nome da entidade*>;
3. O sistema solicita a alteração dos seguintes atributos:
    - <*lista de atributos que podem ser alterados*>
4. O <*nome do ator*> altera os dados desejados e confirma a alteração;
5. O sistema realiza a alteração dos dados informados no passo 4;
6. O sistema exibe uma mensagem de confirmação informando que a alteração do <*nome da entidade*> foi efetivada com sucesso;

Subfluxo Remover <nome da entidade>

1. Este subfluxo inicia quando o <*nome do ator*> solicita remover uma ou mais <*nome da entidade*>;
2. O <*nome do ator*> seleciona quais <*nome da entidade*> deseja remover e solicita a remoção;

3. O sistema solicita a confirmação para a remoção;
4. O <*nome do ator*> confirma a remoção;
5. O sistema remove os <*nome da entidade*> confirmados;
6. O sistema exibe uma mensagem informando que a remoção dos <*nome da entidade*> foi efetivada com sucesso;

Subfluxo Consultar <nome da entidade>

1. Este fluxo inicia quando o <*nome do ator*> solicita consultar <*nome da entidade*>;
2. O sistema solicita o preenchimento dos seguintes filtros:
   - <*lista de filtros*>.
3. O <*nome do ator*> preenche os filtros e solicita a consulta;
4. O sistema apresenta as seguintes informações dos <*nome da entidade*> obtidos na consulta:
   - <*lista de atributos*>.

Validações e regras de negócio

- Esta regra se aplica a todos os subfluxos. Atributos obrigatórios. Se algum atributo obrigatório não tiver sido preenchido, <descrever que ações o sistema deve tomar, por exemplo, "o sistema não completará a operação e notificará ao <*nome do ator*>, solicitando o preenchimento">;
- Esta regra se aplica a todos os subfluxos. Atributos com valores não permitidos. Se algum atributo for preenchido com valor não permitido, <descrever que ações o sistema deve tomar, por exemplo, "o sistema não completará a operação e notificará ao <*nome do ator*>, solicitando o preenchimento">;
- No subfluxo **Remover**, o sistema valida os <*nome da entidade*> selecionados de acordo com as seguintes regras:
  - <*regras de remoção*>.

2.5. Exemplo

Este exemplo apresenta o caso de uso Manter Cliente de uma aplicação de CallCenter.

Incluir Cliente

1. Este subfluxo inicia quando o *Operador de Telemarketing* solicita incluir um *cliente*;
2. O sistema solicita ao *Operador de Telemarketing* o preenchimento dos seguintes atributos:
   - * Nome;
   - * Logradouro. Descreve a rua ou a avenida em que o cliente reside;
   - * Número;
   - * Bairro;
   - * Cidade;

- * Estado (campo de escolha fechada. Valores possíveis: todas os estados cadastrados no sistema);
- CPF;
- Sexo (campo de escolha fechada. Valores possíveis: feminino e masculino).

3. O *Operador de Telemarketing* preenche os atributos acima e confirma a inclusão;
4. O sistema realiza a inclusão dos dados informados pelo *Operador de Telemarketing* no passo 3;
5. O sistema exibe uma mensagem informando que a inclusão do *cliente* foi efetivada com sucesso;

Alterar Cliente

1. Este fluxo inicia quando o *Operador de Telemarketing* solicita alterar um *cliente*;
2. O *Operador de Telemarketing* seleciona um único *cliente*;
3. O sistema solicita a alteração dos atributos listados no passo 2 do subfluxo Incluir.
4. O *Operador de Telemarketing* altera os dados desejados e confirma a alteração;
5. O sistema realiza a alteração dos dados informados no passo 4;
6. O sistema exibe uma mensagem de confirmação informando que a alteração do *cliente* foi efetivada com sucesso;

Remover Cliente

1. Este subfluxo inicia quando o *Operador de Telemarketing* solicita remover um ou mais *clientes*;
2. O *Operador de Telemarketing* seleciona quais *clientes* deseja remover e solicita a remoção;
3. O sistema solicita a confirmação para a remoção;
4. O *Operador de Telemarketing* confirma a remoção;
5. O sistema remove os *clientes* confirmados;
6. O sistema exibe uma mensagem informando que a remoção dos *clientes* foi efetivada com sucesso;

Consultar Cliente

1. Este fluxo inicia quando o *Operador de Telemarketing* solicita consultar *clientes*;
2. O sistema solicita o preenchimento dos seguintes filtros:
   - Nome;
   - CPF.
3. O *Operador de Telemarketing* preenche os filtros e solicita a consulta;
4. O sistema apresenta as seguintes informações dos *clientes* obtidos na consulta:
   - Nome;
   - Logradouro;
   - Número;
   - Bairro;

- Cidade;
- Estado;
- CPF;
- Sexo.

Validações e regras de negócio

- Esta regra se aplica a todos os subfluxos. Atributos obrigatórios. Se algum atributo obrigatório não tiver sido preenchido, o sistema não completará a operação e notificará ao *Operador de Telemarketing*, informando quais campos obrigatórios não foram preenchidos e solicitando o preenchimento dos mesmos;
- Esta regra se aplica a todos os subfluxos. Atributos com valores não permitidos. Se algum atributo for preenchido com valor não permitido, o sistema não completará a operação e notificará ao *Operador de Telemarketing*, informando quais campos foram preenchidos com valores inválidos e solicitando o preenchimento correto;
- No subfluxo **Remover**, o sistema valida os *clientes* selecionados de acordo com as seguintes regras:
    - Cliente que tiver algum chamado em aberto não poderá ser removido.

2.6. Conseqüências

- As operações de manutenção e seus requisitos são documentadas de forma padronizada e estruturada para os diversos tipos de entidade, melhorando o entendimento do comportamento e dos requisitos, facilitando o desenvolvimento de produtos de trabalho das fases seguintes, como por exemplo, análise, projeto e casos de teste;
- As validações e regras de negócio são documentadas de maneira estruturada, evitando omissões e destacando sua importância;
- Os atributos e informações requeridos em cada operação são documentados, facilitando o entendimento da estrutura do sistema e facilitando a modelagem de dados e prototipação de telas;
- Fornece suporte ao conceito de caso de uso definido em [2]: “todo caso de uso deve demonstrar um valor observável”. A solução utiliza o conceito de subfluxos para agrupar em um único caso de uso as operações de Inclusão, Consulta, Alteração e Exclusão.
- Reduz o número de casos de uso do sistema por meio do agrupamento da especificação das operações de manutenção em um único caso de uso, facilitando a gestão dos requisitos.

2.7. Padrões relacionados

- Padrão **Documentação de Atributos**:
    - Utilizado no subfluxo **Inserir** para listar os atributos da entidade; no subfluxo **Alterar**, para descrever os atributos que podem ser alterados; e

no subfluxo **Consultar**, para descrever os filtros e atributos que serão exibidos no resultado da consulta.

3. Documentação de Atributos

3.1. Contexto

Em sistemas de informação, os atributos das entidades possuem diversas características como: nome, descrição, obrigatoriedade, validações, semântica, entre outras. Portanto, a documentação desses atributos deve ser elaborada de forma que essas características não sejam esquecidas.

3.2. Problema

Como definir e documentar de forma padronizada os diversos atributos das entidades, que são informações necessárias durante operações CRUD?

3.3. Forças

- Atributos podem ser de tipos primitivos, enumerados, multivalorados ou de relacionamentos. Os atributos enumerados podem assumir um valor dentro de um domínio fixo de valores. Os atributos de relacionamentos podem assumir como valor uma referência para outras entidades cadastradas no sistema. Os atributos multivalorados podem assumir um ou mais valores referentes a outras entidades cadastradas no sistema.
- Os atributos de entidades podem fazer parte de um conjunto de parâmetros ou filtros de consulta.
- Alguns atributos podem ser opcionais e outros obrigatórios. Atributos obrigatórios devem ter tratamento adequado em caso de não preenchimento na inclusão, alteração ou consulta.
- Se os atributos não foram documentados com as informações necessárias, os seguintes problemas poderão ocorrer: (i) dificuldade na validação dos requisitos com o usuário final por falta de informações sobre os atributos e (ii) inconsistência nos produtos de trabalho gerado nas fases de análise e projeto, implementação e testes.

3.4. Solução

- Documente os atributos como uma lista itemizada associada a uma operação de consulta, inclusão ou alteração. No caso da alteração, se os atributos que podem ser alterados forem os mesmos da inclusão pode-se apenas fazer uma referência aos atributos listados na inclusão.
- Uma descrição breve do atributo deve ser fornecida, quando necessário.
- Marque com um caractere especial os atributos obrigatórios (“*”, por exemplo).
- Para atributos que indicam relacionamento, indique que é um campo de escolha fechada e indique a fonte origem dos dados de escolha. Por exemplo:

Unidade federativa (campo de escolha fechada. Valores possíveis: todas as unidades federativas cadastradas no sistema).

- Para atributos enumerados, indique que é um campo de escolha fechada e indique os valores possíveis. Por exemplo: Sexo (campo de escolha fechada. Valores possíveis: feminino e masculino).
- Para atributos multivalorados, indique que é um campo de escolha múltipla e indique a fonte origem dos dados de escolha.
- Alguns atributos possuem restrição quanto aos valores aceitos. Neste caso, deve-se documentar esta restrição juntamente com o atributo.

3.4.1 Estrutura

- *<atributo>. <descrição do atributo>*
- *<caractere> <atributo obrigatório>*
- *<atributo>* (Campo de escolha fechada. Valores possíveis: *<entidade origem dos dados>*). *<descrição do atributo>*
- *<atributo>* (Campo de escolha fechada. Valores possíveis: *<valor 1>*, *<valor 2>*, ... *<valor n>*). *<descrição do atributo>*
- *<atributo>* (Campo de escolha múltipla. Valores possíveis: *<entidade origem dos dados>*). *<descrição do atributo>*
- *<atributo>. <descrição da validação de valores aceitos>*

3.5. Exemplo

Exemplo de atributo com descrição:

- Logradouro. Descreve a rua ou a avenida em que o cliente reside;

Exemplo de atributo obrigatório:

- * Nome

Exemplo de atributo de relacionamento:

- Estado (campo de escolha fechada. Valores possíveis: todos os estados cadastrados no sistema);

Exemplo de atributo enumerado:

- Sexo (campo de escolha fechada. Valores possíveis: feminino e masculino).

Exemplo de atributo multivalorado:

- Autor do livro (campo de escolha múltipla. Valores possíveis: todos os autores cadastrados no sistema).

Exemplo de atributo com restrição de valores:

- Temperatura corpórea do paciente. Só poderá assumir valor entre 35 e 42 graus.

3.6. Conseqüências

- Os diversos tipos de atributos são documentados de forma simples e padronizada.
- Os atributos obrigatórios são declarados claramente, facilitando sua identificação e tratamento da implementação e testes.

4. Caso de Uso Relatório

4.1. Contexto

Em sistemas de informação, uma grande quantidade de dados é armazenada freqüentemente. Neste contexto, surge a necessidade de visualizar, exportar ou imprimir dados armazenados com o objetivo de conferir, analisar e tomar decisões com base nesses dados.

4.2. Problema

Como documentar os requisitos de relatórios que podem incluir a necessidade de visualizar, exportar ou imprimir dados de entidades de acordo com filtros especificados, agrupamentos, totalizações e informações a serem apresentadas?

4.3. Forças

- O sistema deve permitir extrair dados em diversos formatos (tela, arquivo e impressão).
- O sistema deve tratar a estrutura do relatório, como por exemplo, disposição dos campos, cabeçalho e rodapé, tamanho da fonte e orientação do papel.
- O sistema deve tratar as necessidades para exibição dos dados, como por exemplo, se os dados devem ser agrupados, se devem ser apresentadas totalizações e se existe a necessidade de algum filtro para restringir os dados que serão apresentados.

4.4. Solução

- O **Fluxo básico** descreve que atributos devem ser filtros, quais são de preenchimento obrigatório e quais atributos devem ser exibidos no cabeçalho, corpo ou rodapé.
- O **Fluxo básico** descreve a condição de início. Condições de início indicam os eventos que provocam a execução do caso de uso. Por exemplo: em que situação o relatório deve ser visualizado ou impresso; ou se existe alguma periodicidade requerida.
- Os requisitos especiais são documentados em uma seção independente dos fluxos e subfluxos. Tipicamente, devem ser documentados requisitos de exportação para diversos formatos, regras das seções (regras de agrupamento, cálculo para totalização) e opções de ordenação. Um desenho esquemático do relatório e suas seções pode também ser apresentado. Descrever também o critério para filtro ou extração de dados.

4.4.1 Estrutura

Fluxo básico

1. Este fluxo inicia quando o <*nome do ator*> solicita gerar o relatório <nome do relatório>. <descrever a condição de início do caso de uso>;
2. O sistema solicita o preenchimento dos seguintes filtros:
   - <*lista de filtros*>.
3. Uma vez que o <*nome do ator*> forneça a informação solicitada, uma das seguintes ações é executada:
   - Se o <*nome do ator*> selecionar Imprimir, <descrever ação que deve ser executada>;
   - Se o <*nome do ator*> selecionar Visualizar, <descrever ação que deve ser executada>;
   - Se o <*nome do ator*> selecionar Exportar, <descrever ação que deve ser executada>;
4. O sistema apresenta o resultado na seguinte forma:
   - Cabeçalho. <descrever as informações que devem está contidas no cabeçalho>;
   - Corpo. <descrever as informações que devem estar contidas no corpo, informando lista de atributos, seções de agrupamento, e quebra de seção>;
   - Rodapé. <descrever as informações que devem estar contidas no rodapé>;
   - Totalização. <descrever que totalizações devem ser exibidas>.

Requisitos especiais

- Exportar para diversos formatos. <descrever para que formatos o resultado do relatório deve ser exportado, informando os requisitos necessários para a exportação de cada formato>;
- Regras das seções. <descrever quais são as regras de agrupamento de seções e as regras para o cálculo das totalizações>;
- Opções de ordenação. <listar as opções de ordenação disponíveis e descrever os requisitos para essas ordenações>.
- Regra de extração. <expressão lógica descrevendo como os atributos de filtro e outros critérios devem ser combinados para extrair os dados corretamente>
- Modelo de desenho esquemático:

<Logo><Sistema>	**<Título>**	
<Grupo1>		
	<Campo 1>	<Campo 2>
<Total grupo 1>		**<Soma campo 2>**
		<Página x de y>

4.5. Exemplo

Este exemplo apresenta um relatório de cliente de uma aplicação de CallCenter. O relatório possui totalizações por bairro.

Fluxo básico

1. Este fluxo inicia quando o *Operador de Telemarketing* solicita gerar o relatório de *clientes por bairro*. Este relatório deve ser executado antes da avaliação da carteira de clientes;
2. O sistema solicita o preenchimento dos seguintes filtros:
   - *Código da filial*.
3. Uma vez que o *Operador de Telemarketing* forneça a informação solicitada, uma das seguintes ações é executada:
   - Se o *Operador de Telemarketing* selecionar Imprimir, o sistema deve apresentar a janela de configuração de impressão;
   - Se o *Operador de Telemarketing* selecionar Visualizar, o sistema deve apresentar uma janela com a visualização do relatório;
   - Se o *Operador de Telemarketing* selecionar Exportar, o sistema deve solicitar o tipo de arquivo a ser exportado e gerar o arquivo solicitado conforme padrão definido nos requisitos especiais;
4. O sistema apresenta o resultado na seguinte forma:
   - Cabeçalho. Deve conter o nome do relatório, nome da empresa, nome da filial e a data em que o relatório foi executado;
   - Corpo. Os clientes devem ser agrupados por bairro e as seções devem conter quebras de página a cada bairro. Os seguintes atributos devem ser apresentados: nome do bairro, nome, telefone e data de cadastro do cliente;
   - Rodapé. Deve conter o número da página;
   - Totalização. As totalizações devem ser efetuadas por bairro, apresentando quantos clientes existem em cada bairro.

Requisitos especiais

- Exportar para diversos formatos. Os dados deste relatório devem ser exportados para o Excel, apresentado as informações em colunas;
- Opções de ordenação. O relatório deve ser ordenado por nome do bairro e posteriormente por nome do cliente.
- Regra de extração. Devem ser apresentados no relatório todos os clientes cadastrados no sistema e que são relacionados à filial selecionada no filtro. A identificação da filial encontra-se no cadastro do cliente.
- Modelo de desenho esquemático:

Empresa de Telemarketing **Filial Norte América** **Relatório de clientes por bairro** **01/01/2005**		
Bairro: Varjota		
Nome	Telefone	Data de Cadastro
Gabriela Souza	32678950	01/01/2004
Carlo Pires	29087654	23/04/2004
Total de clientes da Varjota:		**2**
		Página 1

4.6. Conseqüências

- As opções e formatos para extração são descritos.
- A estrutura do relatório e das seções é documentada de forma clara e estruturada.
- Os requisitos para extração da informação são documentados.

4.7. Padrões relacionados

- Padrão **Documentação de Atributos**:
  - Utilizado no **Fluxo básico** para listar os filtros.

5. Caso de Uso Transação

5.1. Contexto

Documentação dos requisitos de operações que são tratadas como um comando atômico que processa várias transações. Tipicamente operações *batch* e operações que requerem apenas um comando de inicio do caso de uso pelo usuário tendo pouca entrada de dados e iteração com o sistema.

5.2. Problema

Como documentar os requisitos de operações que possuem a execução de longa duração ou que são executadas em formato de comando atômico, dando ênfase para os requisitos especiais dessas operações?

5.3. Forças

- Transações que ocorrem freqüentemente em sistemas de informação possuem várias características em comum e é importante que fiquem documentadas de forma uniforme para facilitar o entendimento dos casos de uso.

- O usuário necessita de informação sobre o progresso e o tempo estimado para a conclusão da operação.
- O usuário pode não ter familiaridade com a complexidade da tarefa.
- Transações complexas podem envolver algoritmos e cálculos.
- Durante a operação o usuário pode decidir interrompê-la.

5.4. Solução

- Os requisitos devem documentar a duração média do tempo de execução da operação.
- O **Fluxo básico** descreve que atributos devem ser fornecidos para a execução da operação, indicando quais são obrigatórios.
- O **Fluxo básico** descreve a condição de início. Condições de início indicam os eventos que provocam a execução do caso de uso. Por exemplo, em que situação o caso de uso deverá ser executado ou se existe alguma periodicidade requerida.
- O **Fluxo básico** deve indicar que existe uma opção de cancelamento que pode ser solicitada a qualquer momento.
- Os requisitos especiais descrevem como o progresso da operação será apresentado. O progresso é tipicamente o momento restante para o término, o número das unidades processadas ou a porcentagem do trabalho feita. Tipicamente deve ser fornecido para o usuário o status da execução da operação, informando se a operação ainda está sendo executada, e quanto tempo o usuário necessitará esperar.

5.4.1 Estrutura

Fluxo básico

1. Este fluxo inicia quando o <nome do ator> solicita executar a <nome da transação>. <descrever a condição de início do caso de uso>;
2. O sistema solicita o preenchimento dos seguintes dados:
   - *<lista de atributos de parâmetro para a transação>*.
3. O *<nome do ator>* preenche os dados solicitados no passo 2 e confirma a execução da operação;
4. O sistema executa a operação:
   - <Operações, indicações de algoritmos e de cálculos executados na operação>

Requisitos especiais

- O progresso da operação deverá ser apresentado em <descrever a unidade ou formato em que será apresentado o progresso da operação>.

Regras de negócio

- Descrição de algoritmos e cálculos eventualmente utilizados na operação.

5.5. Exemplo

Este exemplo apresenta o caso de uso Transferir Chamado de um sistema de Call Center. O objetivo deste caso de uso é transferir um chamado de um *Operador de Telemarketing* para outro.

Fluxo básico

1. Este fluxo inicia quando o Operador de Telemarketing solicita transferir um chamado;
2. O sistema solicita o preenchimento dos seguintes dados:
   - * Número dos chamados. (Campo de escolha múltipla);
   - * Nome do novo *Operador de Telemarketing* responsável pelo chamado. (Campo de escolha fechada. Valores possíveis: todos os *Operadores de Telemarketing* ativos cadastrados no sistema). Esse campo deve aparecer em ordem alfabética pelo nome do *Operador de Telemarketing*;
   - * Descrição. Este campo deve conter a descrição do histórico da transferência.
3. O *Operador de Telemarketing* preenche os dados solicitados no passo 2 e confirma a execução da operação;
4. O sistema executa as seguintes operações:
   - Obtém o login do usuário corrente e atribui ao campo responsável pela transferência dos chamados;
   - Obtém a data e hora corrente e atribui ao campo data de criação do histórico;
   - Atribui ao identificador do tipo do histórico o valor “transferência”;
   - A aplicação realiza a transferência dos chamados salvando os dados informados pelo *Operador de Telemarketing* no passo 2 e obtidos pela aplicação no passo 4;

Requisitos especiais

- O progresso da operação deverá ser apresentado em % (percentual) que deverá ser calculado considerando quantos chamados já foram transferidos em relação ao total de chamados selecionado. Por exemplo: o *Operador de Telemarketing* selecionou 10 (dez) chamados para serem transferidos. Quando o sistema estiver efetuado a transferência de 2 (dois) chamados o progresso da operação será 20% (vinte por cento).

Regras de negócio

- Não se aplica.

5.6. Conseqüências

- O retorno sobre o status da execução da transação é fornecido;
- Os passos da transação, algoritmos e cálculos são documentados de forma clara.

5.7. Variantes

- Em transações curtas, o tratamento do progresso da operação pode ser suprimido.

5.8. Padrões relacionados

- Padrão **Documentação de Atributos**:
  - Utilizado no **Fluxo básico** para listar os filtros.

6. Caso de Uso Assistente

6.1. Contexto

Documentação dos requisitos de operações complexas que são executadas em diversos passos, onde decisões ou dados necessitam serem informados em cada passo através da iteração com o usuário.

6.2. Problema

Como documentar os requisitos de uma operação, na qual diversas decisões devem ser tomadas antes que a operação possa ser concluída completamente?

6.3. Forças

- Para concluir a operação, diversos passos precisam ser realizados.
- Um determinado passo pode necessitar ser terminado antes que o passo seguinte possa ser feito.

6.4. Solução

- O **Fluxo básico** descreve o objetivo da operação e quantos passos precisam ser executados.
- O **Fluxo básico** descreve a condição de início. Condições de início indicam os eventos que provocam a execução do caso de uso. Por exemplo: em que situação o caso de uso deverá ser executado ou se existe alguma periodicidade requerida.
- O **Fluxo básico** deve indicar que existe uma opção de cancelamento que pode ser solicitada a qualquer momento.
- Cada **Subfluxo Passo <n>** deve determinar se o usuário não pode começar o passo seguinte antes de terminar o atual.

6.4.1 Estrutura

Fluxo básico

1. O caso de uso inicia quando o <nome do ator> necessita <nome do caso de uso>. <descrever a condição de inicio do caso de uso>;
2. O sistema informa tipicamente o objetivo da operação e quantos passos precisam ser executados;
3. O sistema solicita que o <*nome do ator*> execute o Passo 1;
4. Uma vez que o <*nome do ator*> decida executar o Passo 1, subfluxo **Passo 1** é executado;
5. O caso de uso se encerra.

Subfluxo Passo 1

1. Este subfluxo se inicia quando o <*nome do ator*> solicita <descrever as ações que serão executadas neste passo>;
2. O sistema solicita ao <*nome do ator*> o preenchimento dos seguintes atributos:
   - <*lista de atributos*>.
3. O <*nome do ator*> preenche os atributos;
4. O sistema solicita que o <*nome do ator*> execute o Passo n;
5. Uma vez que o <*nome do ator*> decida executar o Passo <n>, subfluxo Passo <n> é executado;

Subfluxo Passo <n>

1. Este subfluxo se inicia quando o <*nome do ator*> solicita <descrever as ações que serão executadas neste passo>;
2. Para este subfluxo ser executado os subfluxos <*Passo 1, Passo 2, ... Passo n*> devem ter sido executados. Se não existir requisitos de precedência para a execução dos passos, esse item poderá ser omitido;
3. O sistema solicita ao <*nome do ator*> o preenchimento dos seguintes atributos:
   - <*lista de atributos*>.
4. O <*nome do ator*> preenche os atributos;
5. O sistema solicita que o <*nome do ator*> execute o Passo <n+1> ou conclua a operação;
6. Uma vez que o <*nome do ator*> decida executar o Passo <n+1>, subfluxo **Passo <n+1>** é executado;

Subfluxo Passo <final>

1. Este subfluxo se inicia quando o <*nome do ator*> solicita <descrever as ações que serão executadas neste passo>;
2. Para este subfluxo ser executado os subfluxos <*Passo 1, Passo 2, ... Passo n*> devem ter sido executados. Se não existir requisitos de precedência para a execução dos passos, esse item poderá ser omitido;
3. O sistema solicita ao <*nome do ator*> o preenchimento dos seguintes atributos:
   - <*lista de atributos*>.
4. O <*nome do ator*> preenche os atributos;
5. O sistema solicita que o <*nome do ator*> conclua a operação;

6. O caso de uso retorna para o passo 5 do fluxo básico.

6.5. Exemplo

Este exemplo apresenta o caso de uso Submeter Proposta de Seguro de um sistema de administração de seguros para automóveis, que deve ser realizado em três passos. No passo inicial o proponente informa a cidade e o estado onde o veículo irá circular e os dados do veículo. No segundo passo, o sistema apresenta uma lista de coberturas e preços existentes de acordo com os dados informados no passo 1. O proponente seleciona as coberturas desejadas e avança para o passo seguinte. No terceiro e último passo o sistema apresenta o preço total do seguro e solicita a conclusão da operação.

Fluxo básico

1. O caso de uso inicia quando o *Proponente* necessita submeter uma proposta de seguro;
2. O sistema informa que esta operação será executada em 3 passos;
3. O sistema solicita que o *Proponente* execute o Passo 1;
4. Uma vez que o *Proponente* decida executar o Passo 1, subfluxo **Passo 1** é executado;
5. O caso de uso se encerra.

Passo 1

1. Este subfluxo se inicia quando o *Proponente* solicita informar a cidade e o estado onde o veículo irá circular e os dados do veículo;
2. O sistema solicita ao *Proponente* o preenchimento dos seguintes atributos:
   - * *Cidade.* Indica a cidade onde o veículo irá circular;
   - * *Estado.* Indica o onde o veículo irá circular;
   - * *Ano de fabricação do veículo;*
   - * *Ano do modelo do veículo;*
   - * *Modelo do veículo;*
   - * *Marca do veículo.*
3. O *Proponente* preenche os atributos;
4. O sistema solicita que o *Proponente* execute o Passo 2;
5. Uma vez que o *Proponente* decida executar o Passo 2, subfluxo **Passo 2** é executado;

Passo 2

1. Este subfluxo se inicia quando o sistema apresenta uma lista de coberturas e preços existentes;
2. Para este subfluxo ser executado o subfluxo **Passo 1** deve ter sido executado;
3. O sistema apresenta a lista de coberturas e preços existente e solicita ao *Proponente* a seleção das coberturas desejadas;
4. O *Proponente* seleciona as coberturas;
5. O sistema solicita que o *Proponente* execute o Passo 3;
6. Uma vez que o *Proponente* decida executar o Passo 3, subfluxo **Passo 3** é executado;

Passo 3

1. Este subfluxo se inicia quando o *Proponente* solicita a conclusão da operação;
2. Para este subfluxo ser executado os subfluxos **Passo 1** e **Passo 2** devem ter sido executados;
3. O sistema apresenta o preço total do seguro e solicita a conclusão da operação;
4. O *Proponente* conclui a operação;
5. O caso de uso retorna para o passo 5 do fluxo básico.

6.6. Conseqüências

- Organiza e documenta todos os passos que devem ser realizados para concluir uma operação complexa.
- Permite que o usuário possa realizar intervenções, decisões e configurações em estágios intermediários de uma operação complexa.

6.7. Padrões relacionados

- Padrão **Documentação de Atributos**:
  - Utilizado no **Fluxo básico** e subfluxos para listar os atributos.

7. Usos conhecidos

Os padrões apresentados neste artigo têm sido utilizados na fase de elicitação de requisitos em diversos sistemas, tais como um sistema Imobiliário, um sistema de Portal web para administradção e publicação de informações de acervos culturais e um sistema de CallCenter. Porém, por motivos de confidencialidade, mais detalhes dos usos conhecidos não podem ser fornecidos.

Em [2], os exemplos de casos de uso CRUD seguem estrutura similar a proposta no padrão **Caso de Uso CRUD**.

8. Agradecimentos

Este trabalho foi suportado pelo Instituto Atlântico.

Os autores agradecem aos responsáveis pelo processo de revisão, em especial a Rosana Teresinha Vaccare Braga, pelas contribuições realizadas no aprimoramento do artigo.

Referências

[1] Chrissis, M. B., Konrad, M., Shrum, S. **CMMI Guidelines for Process Integration and Product Improvement**. Addison-Wesley, 2004.

[2] Rational Unified Process®, Version 2002.05.00. Rational Software Corporation, 2001.

[3] COCKBURN, A. **Writing effective: use cases**. Addison-Wesley Boston, 2001.

[4] SCHNEIDER, G.; WINTERS, J. **Applying Use Case: A Practical Guide**. 2nd ed. Addison-Wesley, 2001.

[5] KRUCHTEN, P. **The Rational Unified Process: an introduction**. Addison-Wesley, 2001.

[6] BITTNER, K., SPENCE, I. **Use Case Modeling**. Addison Wesley, 2002

[7] JACOBSEN; CHRISTERSON; OVERGAARD. **Object-oriented software engineering: a use case-driven approach**. Addison-Wesley, 1992.

Patterns for Secure Operating System Architectures

Eduardo B. Fernandez and Tami Sorgente
Dept. of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL
{ed, tami@cse.fau.edu}

Abstract

An operating system (OS) interacts with the hardware and supports the execution of all the applications. As a result, its security is very critical. Most of the reported attacks occur through the OS. The security of individual execution time actions such as process creation and memory protection is very important and we have previously presented patterns for these functions. However, the general architecture of the OS is also very important. We present here patterns for the four basic OS architectures and evaluate their use in different environments. We consider general aspects but we emphasize those aspects that affect security.

1 Introduction

Operating systems (OS) act as an intermediary between the user of a computer and its hardware. The purpose of an OS is to provide an environment in which users can execute programs in convenient and efficient manner [Sil05]. OSs control and coordinate the available resources to present to the user an abstract machine with convenient features. The architecture of the OS organizes components to structure its functional and non-functional aspects. The OS is the most critical of the software layers because compromise can affect all applications and persistent data. Most of the reported attacks occur through the OS [Fer01a]. The security of individual execution time actions such as process creation and memory protection is very important and we have presented patterns for these functions [Fer02, Fer03]. However, the general architecture of the OS is also very important for the ability of the system to provide a secure execution environment.

We present here patterns to help a designer select an architecture according to the security requirements of the applications. Other aspects such as performance, reliability, or real-time properties can also affect the choice. While we mention some of those properties, we focus here on security aspects.

Most operating systems use one of five basic architectures [Sil05, Tan01]. One of them, the *monolithic* architecture has little value for security and it is only mentioned as a possible variant of the modular architecture. We present here patterns representing an abstract view of the other four architectures from the point of view of security The first pattern is the *Modular Operating System Architecture* which describes the components of the operating system as communicating object-oriented modules. In the *Layered Operating System Architecture*, the OS components are assigned to a set of hierarchical layers. The *Microkernel Operating System Architecture* assigns all its functions to servers that communicate through a common module. The *Virtual Machine Operating System Architecture* provides virtual copies

of the underlying hardware that can be used for execution of different OSs. Figure 1 describes a pattern diagram of the OS architectures, indicating their relationships. The figure shows also two variants of the basic patterns, corresponding to common combinations of the three basic architectures.

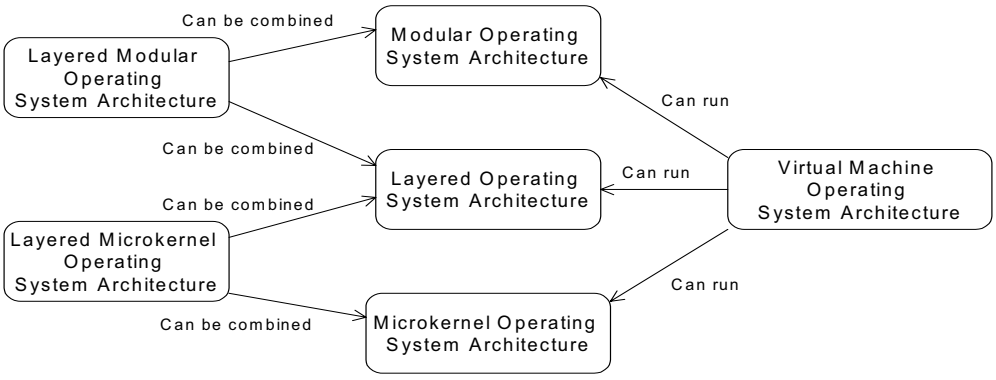


Figure 1. Pattern diagram of OS Architectures

Opearting system functionality can be divided between the kernel (or OS proper), components and the user applications or utilities. Typically, the kernel of an OS includes the following functional units:

- *Process Management*- handles creation and deletion of processes, communication and scheduling.
- *Memory Management*- keeps track of which parts of memory are used by which processes, allocates and deallocates memory.
- *File Management*- handles creation and deletion of files and directories, file searches, and mapping files to secondary storage
- *I/O Management*- provides interfaces to hardware device drivers , as well as handling mass memory management components including buffering, caching, and spooling.
- *Networking*- controls communication path between two or more systems.
- *Protection System*- includes authentication of users and file and memory protection.

In addition, the OS includes a *User Interface,* which communicates between user and OS through command interpreters, and a variety of utilities. These units may be further divided for specific applications. In [Fer02] and [Fer03] we presented patterns for the security of some of these functions. The four architectures we consider show how the structure of the functional units affect security.

2 Modular Operating System Architecture

The OS services can be separated into modules each representing a basic function or component. The core module of the kernel is always in memory, has the necessary functionality to start itself, and the ability to load other modules. Modules are loaded on demand when needed. Each module performs a function and may take parameters.

2.1 Example

Our group is building a new OS that should support various types of devices requiring dynamic services with a large variety of security requirements. We want to dynamically add OS components, functions, and services, as well as tailor their security aspects according to the type of application. For example, a media player may require support to prevent copying of the contents. We need a very flexible architecture. .

2.2 Context

A variety of applications with diverse security requirements, but where the requirements are not very strict.

2.3 Problem

We need to be able to add/ remove functions in the easiest way so we can accommodate applications with a variety of security requirements. How do we structure the functions for this purpose?

The possible solution is constrained by the following forces:

- OSs for PCs and other types of platforms require a large variety of plug-ins. New plug-ins appear frequently and we need the ability to add and remove them in a convenient way.
- Some of the plug-ins may contain malware and we need to isolate their execution so they do not affect other processes.
- We would like to hide security-critical modules from the direct visibility of other modules to avoid possible attacks.
- For performance and flexibility, active (loaded) modules can call each other, which is a possible source of attacks.

2.4 Solution

Define a core module that can dynamically load and link modules as needed. By loading only needed modules we can restrict visibility. We can also have different versions of the modules with different degrees of security and load them according to application security requirements. Critical modules can execute in their own process/thread for better isolation but this may restrict flexibility. Calls between modules can be checked.

Structure

Figure 2 shows a class diagram for this pattern. The KernelCore is the core of the Modular OS. A set of LoadableModules is associated with the KernelCore, indicating the modules that could be loaded. Any LoadableModule can call any other LoadableModule.

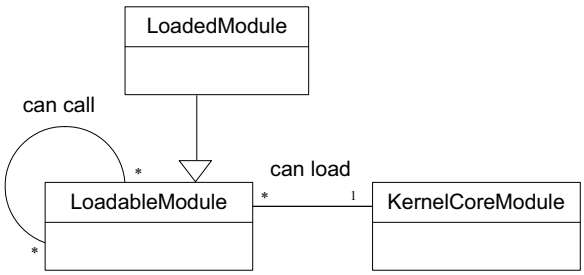


Figure 2. Class diagram for the Modular Operating System Architecture pattern

2.5 Implementation

– Separate the functions of the OS into independent modules according to whether:
  - They are complete functional units.
  - They are critical with respect to security.
  - They should execute in their own process for security reasons or thread for performance reasons.
  - They should be isolated during execution because they may contain malware.
– Define a set of loadable modules. New modules are later added at this point.
– Define a communication structure for the resultant modules. Operations should have well defined call signatures and all calls should be checked. To prevent incorrect commands or malformed parameters.
– Define a preferred order for loading some basic modules. Modules that are critical for security should be loaded only when needed to reduce their exposure to attacks.

2.6 Example resolved

We structured the functions of our system following the Modular Architecture pattern. Because each module could have its own address space, we can isolate its execution. Because each module can be designed independently, they can have different security constraints in their structure. This structure gives us flexibility with a reasonable degree of security.

2.7 Variants

Monolithic kernel. In this case the operating system is a collection of procedures. Each procedure has a well defined interface in terms of parameters and results and each one is free to call any other one [Tan01]. There is no structure between operating system, components, services, and user applications. The difference between monolithic and modular is that in the monolithic approach, all the modules are loaded together at installation time, instead of being brought in on demand. As indicated earlier, this approach is not very attractive for secure systems.

2.8 Known uses

The Solaris 10 Operating System (Figure 3) is designed in this way. Its kernel is dynamic and composed of a core system that is always resident in memory [Sun04]. The types of Solaris 10 loadable modules are represented in Figure 3 as loaded by the kernel core. This diagram does not represent the communication links between

individual modules. Another example is ExtremeWare from Extreme Networks [Ext]. Some versions of Linux are somewhat in between modular and monolithic, in that some modules can be loaded when needed.

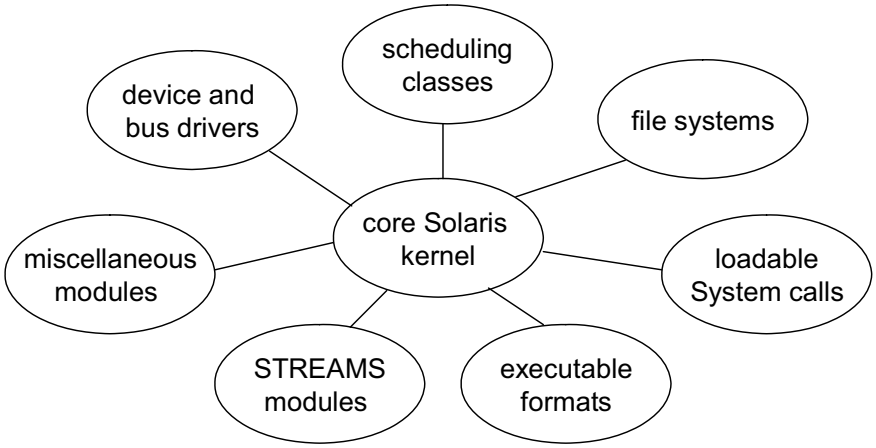


Figure 3. The modular design of the Solaris 10 Operating System [Sil03]

2.9 Consequences

The Modular Operating System Architecture Pattern has the following advantages:

- The flexibility to add/ remove functions contributes to security in that we can add new versions of modules with better security.
- Each module is separate and communicates with other modules over known interfaces. We can introduce controls in these interfaces.
- It is possible to partially hide critical modules by loading them only when needed and removing them after use.
- By giving each executing module its own address space we can isolate the effects of a rogue module.

The Modular Operating System Architecture Pattern has the following liabilities:

- Any module can see all the others and potentially interfere with their execution.
- Uniformity of call interfaces between modules makes it difficult to apply stronger security restrictions to critical modules.

2.10 Related patterns

The Controlled Execution Environment pattern [Fer02] can be used to isolate executing modules.

3 The Layered Operating System Architecture

The overall features and functionality of the OS are decomposed and assigned to hierarchical layers. This provides clearly defined interfaces between each section of the operating system and between user applications and the OS functions. Layer i uses services of a lower layer i-1 and does not know the existence of a higher layer i+1.

3.1 Example

Our team is now handling an OS to support very complex applications. Complexity brings along vulnerability so we need a way to separate concerns. We also want to control the calls between OS components and services to improve security and reliability. Finally, we would like to permanently hide critical modules. We tried a modular architecture but it did not have enough structure to do all this systematically and does not allow us to hide modules permanently.

3.2 Context

A variety of complex applications with diverse and stringent security requirements. Flexibility is not an important concern.

3.3 Problem

Complex applications require separation of concerns for better understanding, errors lead to security flaws. Unstructured modules as in modular architectures have the problem that all modules know about the existence of all other modules, which facilitates attacks. N many systems a good part of the units are stable and only some of them need to be replaced.

The possible solution is constrained by the following forces:

- Interfaces should be stable and well defined. Going through any interface could imply authorization checks.
- Parts of the system should be exchangeable or removable without affecting the rest of the system. For example, we could replace some parts of the system when we need more security.
- Similar responsibilities should be grouped to help understandability and maintainability. This contributes indirectly to improve security.
- We should control module visibility to avoid possible attacks from other modules.
- Complex units need further decomposition. This makes the design simpler and clearer and also improves security.

3.4 Solution

Define a hierarchical set of layers and assign functional components (units) to each layer. Each layer presents an abstract machine to the layer above it, hiding implementation details of the lower layers. Now we can hide modules by placing them in the lower levels. Each level defines a set of services to the level above, these services can apply security checks when invoked. A whole lower level can be replaced by a more secure version.

Structure

Figure 4 shows a class diagram for the Layered Operating System Architecture pattern. Layer N represents the highest level of abstraction, and Layer 1 is the lowest level of abstraction. The main structural characteristic is that the services of Layer i are used only by Layer i + 1. Each layer may contain complex entities consisting of different units.

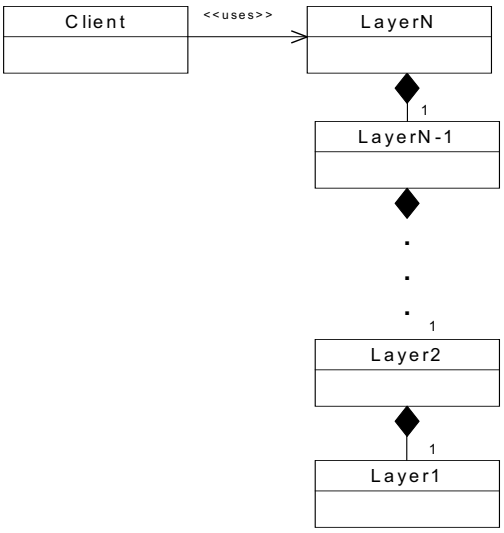


Figure 4. Class diagram for Layered Operating System Architecture pattern

Dynamics

In Figure 5, a user (at the application level) wishes to open a file located in a block of a disk (at a lower level):

- A user sends an openFile request to the OSInterface
- The OSInterface interprets the openFile request.
- The openFile request is sent from the OSInterface to the FileManager
- The FileManager sends a readBlock request to the DiskDriver

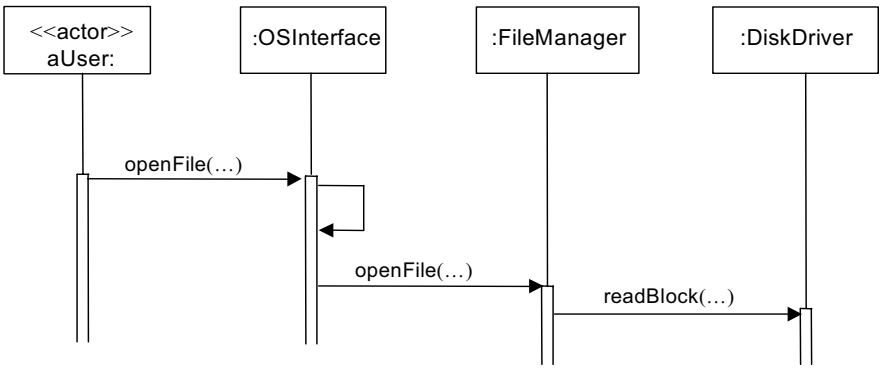


Figure 5. Sequence diagram for opening and reading a disk file

3.5 Implementation

- List all units in the system and define their dependencies.
- Assign units to levels such that units in higher levels depend only on units of lower levels. This may require a new unit decomposition.
- Once the modules in a given level are assigned, define a language (interface) for this level. This language includes the operations that we want to make visible to the next level above. Add well-defined operation signatures and security checks in these operations to assure the proper use of the level.
- Hide in lower levels those modules that control critical security functions (this will prevent direct attacks).

3.6 Example resolved

We structured the functions of our system as in Figure 6 and now we have a way to control interactions and enforce abstraction. For example, the file system can use the operations of the disk drives and enforce similar restrictions in the storage of data. The user of the file cannot take advantage of the implementation details of the disk driver to attack the system.

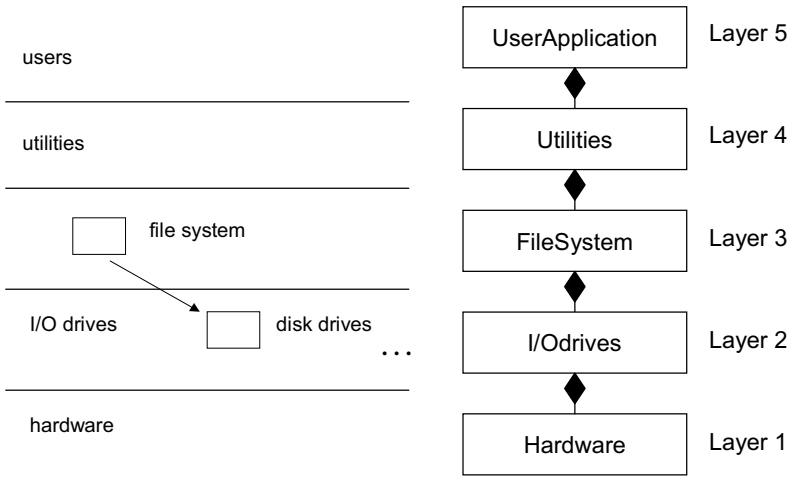


Figure 6. An example of the use of a Layered OS architecture.

3.7 Variants

Layered modules. The modules of the Modular architecture are assigned to different layers. Now, in addition to visibility of modules due to activation, we have visibility constraints due to layering. This could improve the security of the Modular architecture.

Layer skipping. In this architecture there are special applications able to skip layers for added performance (going directly to another layer reduces call overhead). This structure implies a tradeoff between performance and security. By deviating from the strict hierarchy of the layered system, there may not be enforcement of security policies between layers for these applications.

3.8 Known uses

The Symbian OS (Figure 7) uses a variation of the layered approach [Sym01].

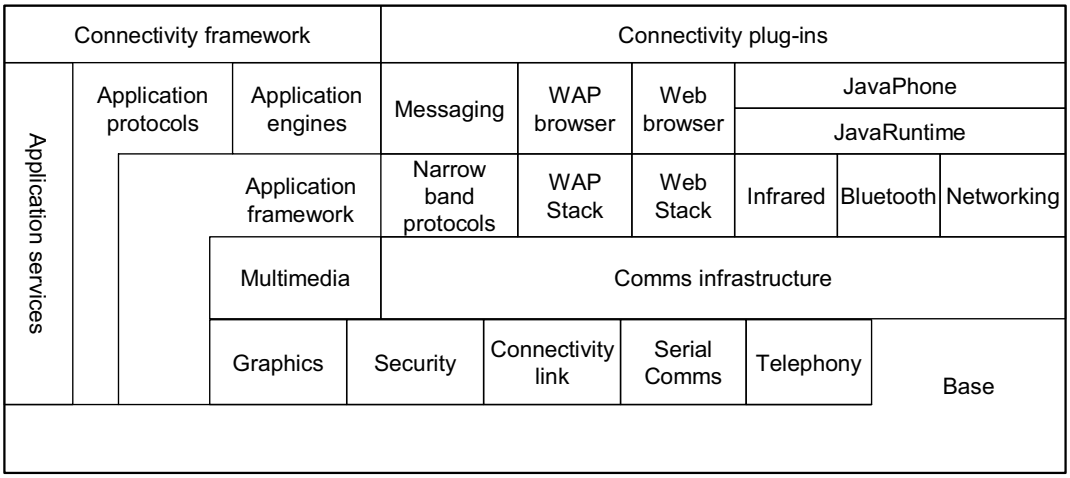


Figure 7. Symbian OS Layered Architecture [Sym01]

The UNIX operating system (Figure 8) is separated into 4 layers with clear interfaces between the system calls to the kernel and between the kernel and the hardware.

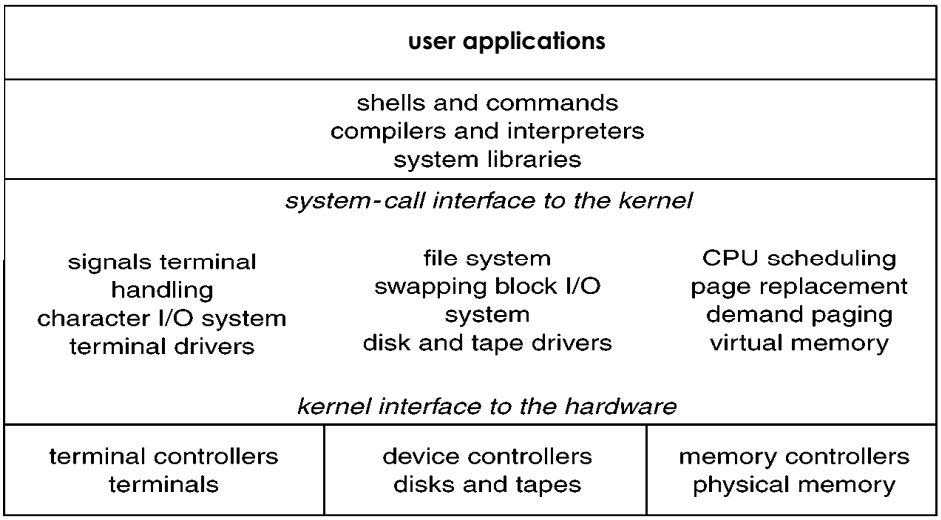


Figure 8. UNIX OS Layered Architecture (from [Sil05])

IBM's OS/2 also uses this approach [OS2].

3.9 Consequences

The Layered Operating System Architecture Pattern has the following advantages:

- Lower levels can be changed without affecting higher layers. We can add or remove security functions in the lower levels as needed.
- Clearly defined interfaces between each OS layer and the user applications improve security.
- Control of information using layered hierarchical rules, using enforcement of security policies between layers.
- The fact that layers hide implementation aspects is useful for security in that possible attackers cannot exploit lower level details.

The Layered Operating System Architecture Pattern has the following liabilities:

- It may not be clear what to put in each layer; in particular related modules may be hard to allocate. There may be conflicts between functional and security needs when allocating modules.
- Performance may decrease due to the indirection of calls through several layers. If we try to improve performance we may sacrifice security.

3.10 Related patterns

This pattern is a specialization of the Layers architectural pattern [Bus96]. Security versions of the Layers pattern have appeared in [Fer02] and in [Yod97].

4 The Microkernel Operating System Architecture

Move as much as possible of the OS functionality from the kernel and put it in specialized servers, coordinated by a microkernel. The microkernel itself has a very basic set of functions. OS functional components and services are implemented as external and internal servers.

4.1 Example

We are building an OS for financial applications. This implies a range of applications with different reliability and security requirements (some are very critical) and a variety of plug-ins. We would like to provide OS versions with different types of modules, some more secure, some less so.

4.2 Context

A variety of applications with diverse security requirements. Some of these applications may be very sensitive and are constantly changing. The platform itself may also change frequently.

4.3 Problem

In general purpose environments we need to be able to add new functionality with variation in security and other requirements as well as provide alternative implementations of services to accommodate different application requirements.

The possible solution is constrained by the following forces:

- The application platform must cope with continuous hardware and software evolution; these additions may have very different security or reliability requirements.
- Strong security or reliability requirements indicate the need for modules with well-defined interfaces.
- We may want to perform different types of security checks in different modules, depending on their security criticality.
- We would like a minimum of functionality in the kernel so we have a minimum of processes running in supervisor mode. A simple kernel can be checked and this is good for security.

4.4 Solution

The Microkernel is the central communication for the OS. Separate all functionality into specialized services with well-defined interfaces and provide an efficient way to route requests to the appropriate servers. Each server can be built with different security constraints. The Microkernel mostly routes requests to servers and has minimal functionality.

Structure

There is one **Microkernel** and several **internal** and **external servers**, each providing a set of specialized services (Figure 9). In addition to the servers, an **Adapter** is used between the **Client** and the microkernel or an external server. The Microkernel controls the internal servers.

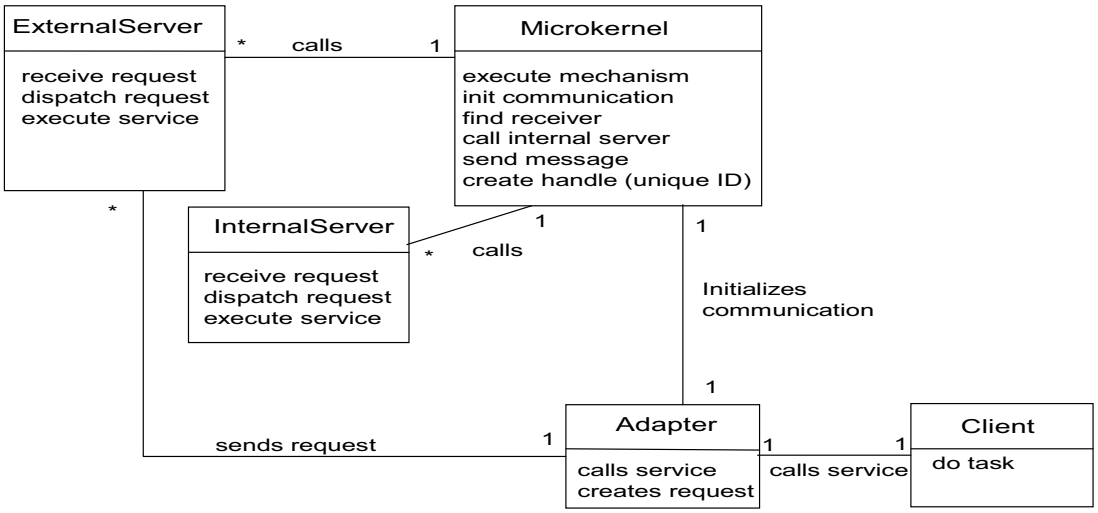


Figure 9. Class diagram for the Microkernel Operating System Architecture pattern

Dynamics

A client requests a service from an external server using the following sequence (Figure 10):

- The adapter receives the request and asks the microkernel for a communication link with the external server.
- The microkernel checks for authorization to use the server, determines the physical address of the external server and returns it to the adapter
- The adapter establishes a direct communication link with the external server.
- The adapter sends the request to the external server using a procedure call or a remote procedure call (RPC). The RPC can be checked for well-formed commands, correct size and type of parameters (we can check signatures).
- The external server receives the request, unpacks the message and delegates the task to one of its own methods. All results are sent back to the adapter.
- The adapter returns to the client, which in turn continues with its control flow.

4.5 Implementation

- Identify the core functionality necessary for implementing external servers and their security constraints. Typically, basic functions of the OS should be internal servers, utilities, or user-defined services should go into external servers. Each server can use the patterns from [Fer02] and [Fer03] for their secure construction.
- Define policies to restrict access to external and internal servers. Clients may be allowed to call only some specific servers.
- Find a complete set of operations and abstractions for every category identified.

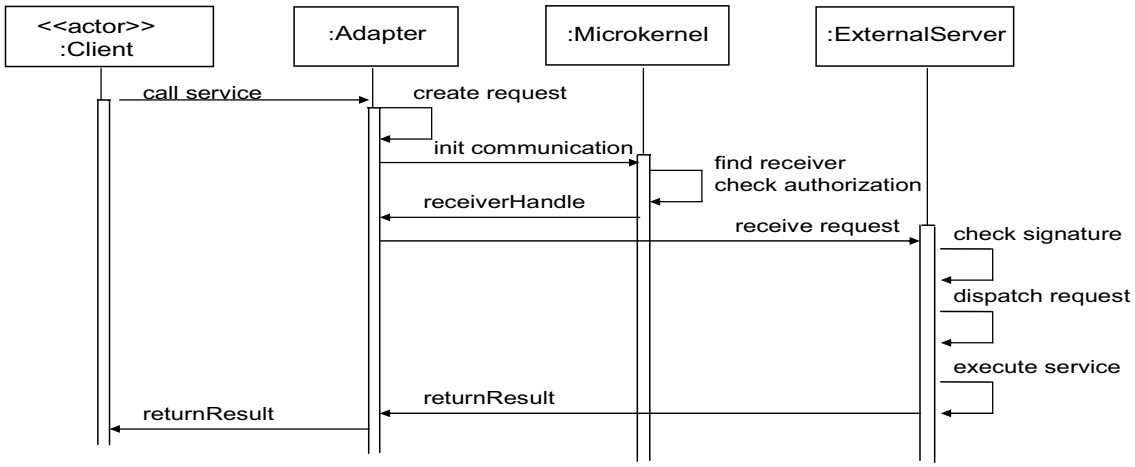


Figure 10. Sequence diagram for performing an OS call through the microkernel

- Determine strategies for request transmission and retrieval.
- Structure the microkernel component. The microkernel should be simple enough to make sure of its security properties (no malware for example).
- Design and implement the internal servers as separate processes or shared libraries. Add security checks in each server.
- Implement the external servers. Add security checks in each service provided by the servers.

4.6 Example resolved

By implementing our system using a microkernel we can have several versions of each service, each with different degrees of security and reliability. We can replace servers dynamically if needed. We can also control access to specific servers and make sure that they are called in the proper way.

4.7 Variants

Layered Microkernel. The Microkernel OS Architecture Pattern can be combined with the Layered OS Architecture pattern. In this case, servers can be assigned to levels and a call is accepted only if it comes from a level above the server level.

4.8 Known uses

The PalmOS Cobalt (Figure 11). This OS has a preemptive multitasking kernel that provides basic task management. Many applications in the PalmOS do not use the microkernel services; they are handled automatically by the system. The microkernel functionality is provided for internal use by system software or for certain special purpose applications [PalmOS].

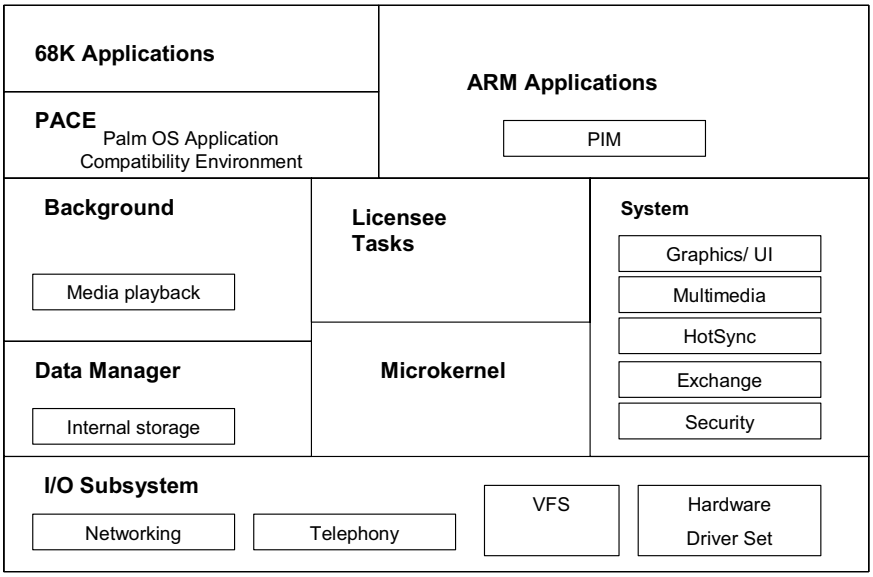


Figure 11. PalmOS Microkernel combined with Layered OS Architecture [PalmOS].

The QNX Microkernel (Figure 12) is intended mostly for communication and process scheduling [QNX].

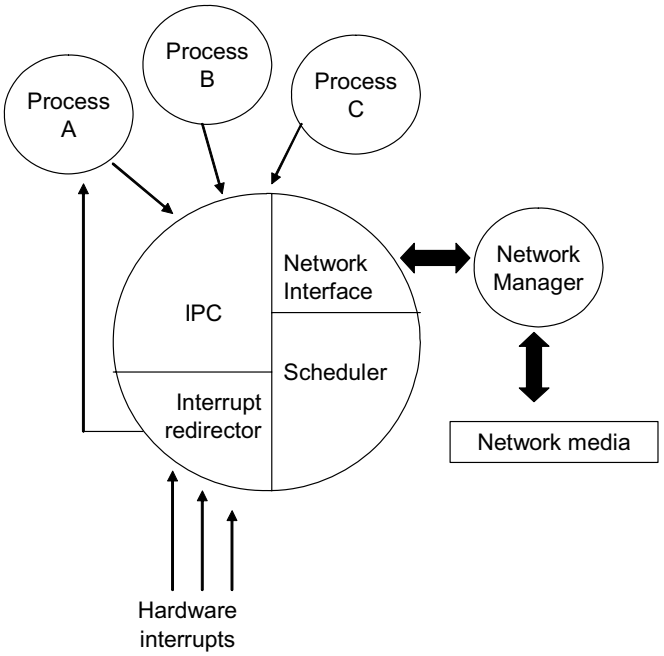


Figure 12. QNX Microkernel Architecture [QNX]

Mach and Windows NT also use some form of microkernels [Sil05].

4.9 Consequences

The Microkernel Operating System Architecture Pattern has the following advantages:

- Flexibility and extensibility – if you need an additional function or an existing function with different security requirements you only need to add an external server. Extending the system capabilities or requirements only require addition or extension of internal servers.
- The Microkernel mediates all calls for services and can apply authorization checks. In fact, the microkernel is in effect, a concrete realization of a reference monitor [Fer01].
- The well-defined interfaces between servers allow each server to check each request for its services.
- Can add even more security by putting fundamental functions in internal servers.
- Servers usually run in user mode, which further increases security.
- The microkernel is very small and can be verified or checked for security.

The Microkernel Operating System Architecture Pattern has the following liabilities:

- Communication overhead since all requests go through the Microkernel.
- Some extra complexity.

4.10 Related patterns

This pattern is a specialization of the microkernel pattern [Bus96]. As indicated, the microkernel itself is a concrete version of the Reference Monitor [Fer01]. The Adapter is an example of the Adapter pattern {Bus96].

5 The Virtual Machine Operating System Architecture

Provides a set of replicas of the hardware architecture (Virtual Machines), which can be used to execute (maybe different) operating systems with a strong isolation between them.

5.1 Example

A web server is hosting applications for two competing companies. These companies use different operating systems. We want to ensure that neither of them can access the other company's files or launch attacks against the other system.

5.2 Context

Mutually suspicious sets of applications that need to execute in the same hardware. Each set requires isolation from the other sets.

5.3 Problem

Sometimes we need to execute different operating systems in the same hardware. How do we keep those operating systems isolated from each other in such a way that their executions don't interfere with each other?

The possible solution is constrained by the following forces:

- Each OS needs to have access to a complete set of hardware features to support its execution.
- Each OS has its own set of machine dependent features, e.g., interrupt handlers. In other words, each OS uses the hardware in different ways.
- When an OS crashes or it is penetrated by a hacker, the effects of this situation should not propagate to other OSs in the same hardware.
- There should be no way for a malicious user in a VM to get access to the data or functions of another VM.

5.4 Solution

Define an architectural layer that is in control of the hardware and supervises and coordinates the execution of each OS environment. This extra layer, usually called a Virtual Machine Monitor (VMM) or Hypervisor presents to each operating system a replica of the hardware. The VMM intercepts all system calls and interprets them according to the originating OS.

Structure

Figure 13 shows a class diagram for the Virtual Machine Operating System Architecture (VMOS). The VMOS contains one **VirtualMachineMonitor** and multiple **Virtual Machines** (VM). Each VM can run a **Local Operating System** (LocalOS). The Hypervisor supports each LocalOS and is able to interpret its system calls. As a **LocalProcess** runs on a LocalOS the VM passes the OS system calls to the Hypervisor, which executes them in the hardware.

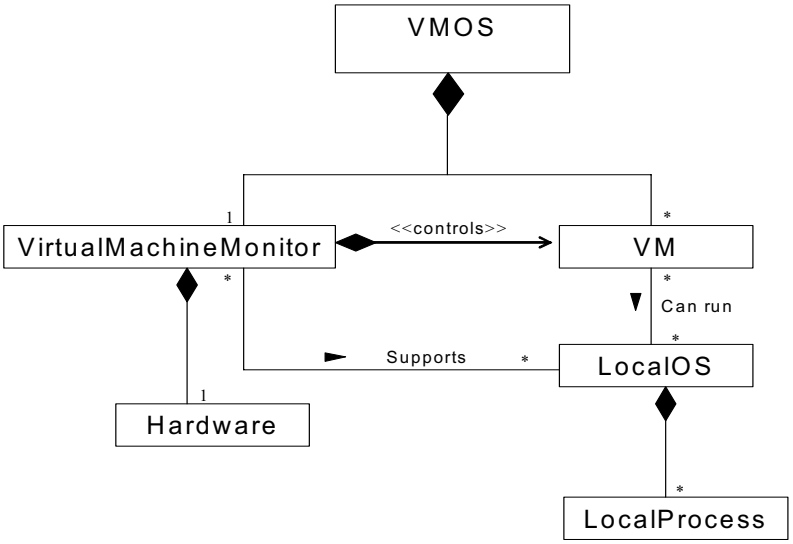


Figure 13. Class diagram for the Virtual Machine Operating System pattern

Dynamics
In Figure 14 a local process wishing to perform a system operation uses the following sequence:

- A LocalProcess makes an OS call to the LocalOS.
- The LocalOS maps the OS call to the VMM (by executing a privileged operation).
- The VMM interprets the call according to the originating OS from where it came and it executes the operation in the hardware.
- The VMM sends return codes to the LocalOS to indicate successful instruction execution as well as results of the instruction execution.
- The LocalOS sends the return code and data to the LocalProcess.

5.5 Implementation

- Select the hardware that will be virtualized. All of its privileged instructions must trap when executed in user mode (this is the usual way to intercept system calls).
- Define a representation (data structure) for describing OS features that map to hardware aspects, e.g. meaning of interrupts, disk space distribution, etc. and build tables for each operating system to be supported. .
- Enumerate the system calls for each supported OS and associate them with specific hardware instructions.

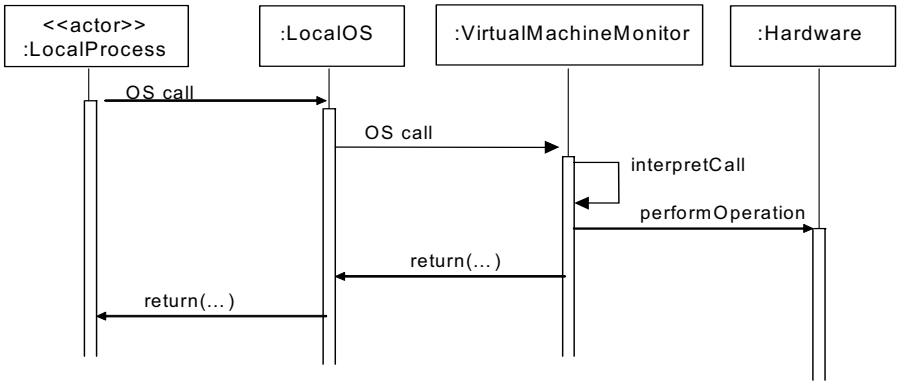


Figure 14. Sequence diagram for performing an OS call on a virtual machine

5.6 Example resolved

In the example of Figure 15, two companies using Windows and Linux can execute their applications in different virtual machines. The VMM provides a strong isolation between these two execution environments.

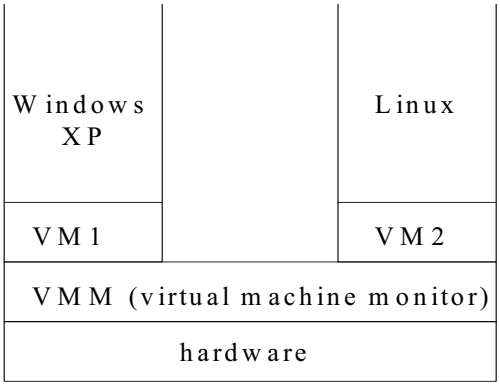


Figure 15. Virtual Machine OS example

5.7 Variants

This architecture is orthogonal to the other three architectures discussed earlier and can execute any of them as local operating systems.

KVM/370 was a secure extension of VM/370 [Gol79]. This system included a formally verified security kernel and its VMs executed in different security levels, e.g. top secret, confidential, etc. In addition to the isolation provided by the VMM, this system also applied the multilevel model secure flow control.

5.8 Known uses

- IBM VM/370 [Cre81]. This was the first VMOS, it provided VMs for an IBM370 mainframe.
- VMware [Nie00]. This is a current system that provides VMs for Intel x86 hardware.
- Solaris10 [Sun04] calls the VMs "containers" and one or more applications execute in each container.
- Connectix [Con] produces virtual PCs to run Windows and other operating systems.
- Xen is a VMM for the Intel x86 developed as a project at the University of Cambridge, UK [Bar00].

5.9 Consequences

The Virtual Machine Operating System Architecture Pattern has the following advantages:

- The VMM intercepts and checks all system calls. The VMM is in effect a Reference Monitor [Fer01] and provides total mediation on the use of the hardware. This can provide a strong isolation between virtual machines [Ros05].
- Each environment (VM) does not know about the other VM(s), this helps prevent cross-VM attacks.
- There is a well-defined interface between the VMM and the virtual machines.
- The VMM is small and simple and can be checked for security.

The Virtual Machine Operating System Architecture Pattern has the following liabilities:

- All the VMs are treated equally. If one needs virtual machines with different levels of security, it is necessary to build specialized versions as done in KVM370 (see Variants).
- Extra overhead in use of privileged instructions.
- It is rather complex to let VMs communicate with each other (if this is needed).

5.10 Related patterns

- Reference Monitor [Fer01]. As indicated, the VMM is a concrete version of a Reference Monitor.
- The operating system patterns in [Fer02] and [Fer03] can be used to implement the structure of a VMOS.

Acknowledgements

This work was supported by a grant from the US Dept. of Defense (DISA), administered by Pragmatics, Inc**.** The comments of our shepherd, Raphael Y. de Camargo, were very useful in improving this paper. Finally, further improvements came from the members of the Writers' Workshop at SugarLoafPLoP 2005.

References

[Bar00] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization", *Procs. of the ACM Symp. on Operating System Principles, SOSP'03.*

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns,* Volume 1. Wiley, 1996.

[Con] Connectix Corp., "The technology of virtual machines", white paper, San Mateo, CA, http://www.connectix.com

[Cre81] R. J. Creasy, "The origin of the VM/370 Time-Sharing System", *IBM Journal of Research and Dev*., vol. 25, No 5, 1981, 483-490.

[Ext] Extreme Networks, http://www.extremenetworks.com/products/OS/

[Fer01] E.B.Fernandez and R. Pan "A pattern language for security models", http://jerry.cs.uiuc.edu/~plop/plop2001/accepted_submissions/

[Fer02] E.B.Fernandez, "Patterns for operating systems access control", *Procs. of PLoP 2002*, http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings.html

[Fer03] E. B. Fernandez and J. C. Sinibaldi, "More patterns for operating system access control", *Proc. of the 8th European conference on Pattern Languages of Programs, EuroPLoP 2003*, http://hillside.net/europlop, 381-398.

[Gam95] E. Gamma, R. Helm,R. Johnson, and J. Vlissides, *Design patterns – Elements of reusable object-oriented software*, Addison-Wesley 1995.

[Gol79] B.D. Gold, R.R. Linde, R.J. Peeler, M. Schaefer, J.F. Scheid, and P.D. Ward, "A security retrofit of VM/370", *Procs. of the Nat. Comp. Conf. (NCC 1979),* 335-344.

[Har02] H. Hartig, "Security Architectures Revisited", *Proceedings of the 10th ACM SIGOPS European Workshop (EW 2002), September 22—25 2002, Saint-Emilion, France,* http://os.inf.tu-dresden.de/papers_ps/secarch.pdf

[Nie00] "Examining VMware", *Dr. Dobbs Journal*, August 2000, 70-76.

[OS2] http://www-306.ibm.com/software/os/warp/

[Pfl03] C.P.Pfleeger, *Security in computing,* 3rd Ed., Prentice-Hall, 2003. http://www.prenhall.com

[Pri04] T. Priebe, E.B.Fernandez, J.I.Mehlau, and G. Pernul, "A pattern system for access control ", in *Research Directions in Data and Applications Security XVIII,* C. Farkas and P. Samarati (Eds.), Procs of the 18th. Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Sitges, Spain, July 25-28, 2004, 235-249.

[PalmOS] http://www.palmos.com/dev/tech/overview.html

[Phi03] Philips, "Current Trends in Operating System kernels", July 2003. http://db.ilug-bom.org.in/lug-authors/philip/docs/os-tech.html

[QNX] QNX Software systems, http://www.qnx.com

[Ros05] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends", *Computer*, IEEE May 2005, 39-47.

[Sch00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software*
architecture, vol. 2 , *Patterns for concurrent and networked objects,* J. Wiley & Sons, 2000.

[Sha02] J.S.Shapiro and N. Hardy, "EROS: A principle-driven operating system from the ground up", *IEEE Software*, Jan./Feb. 2002, 26-33. See also: http://www.eros-os.org

[Sil05] A. Silberschatz, P. Galvin, G. Gagne, *Operating System Concepts (7th Ed.),* John Wiley & Sons, 2003.

[Sun04] http://www.sun.com/software/whitepapers/solaris10/s10security.pdf

[Sym01] http://www.symbian.com/developer/

[Tan01] A. Tanenbaum, *Modern Operating Systems (2nd Ed.),* Prentice Hall, 2001.

[Yod97] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security". *Procs. PLOP'97*, http://jerry.cs.uiuc.edu/~plop/plop97 Also Chapter 15 in *Pattern Languages of Program Design*, vol. 4 (N. Harrison, B. Foote, and H. Rohnert, Eds.), Addison-Wesley, 2000.

Architectural Patterns to Secure Applications with an Aspect Oriented Approach.

Christian Paz-Trillo[1], **Vladimir Rocha**[1]

[1]Department of Computer Science – Institute of Mathematics and Statistics
University of São Paulo
Rua do Matão, 1010 – 05508-090 São Paulo, SP

{cpaz, vmoreira}@ime.usp.br

Abstract. *Today, security problems involving software are serious. Most security problems are caused by attacks through the so-called "security holes". Security holes usually appears because given that security is a crosscutting concern, its Object-Oriented implementation results in systems tough to understand, difficult to evolve and with intruder code in application domain classes. Aspect Orientation is a technique created to deal with this kind of crosscutting concerns, and there is some work using it for security implementation. We present two security patterns with an Aspect-Oriented approach and show how they interact with the application layer. These patterns handle functionality access and put on the security issues on the architectural application level.*

1. Introduction

The variety and increasing number of system attacks have generated serious problems involving from subjective issues, as trust in the system, to objective issues as data integrity and privacy [Redwine and Davis, 2004]. Recently, in the software engineering area, it has been recognized that the majority of these attacks are addressed to the so-called "security holes" of a system. The significant characteristic of all these systems is that security matters were considered only in latest phases of software development process [Halkidis et al., 2004].

When security fixes are done in latest stages of the development process, fixed vulnerabilities can remain hidden for all previous stages, making it difficult to identify the security holes. For example, in the implementation phase it was necessary to add a method providing data encryption and this method which was not specified in the system architecture. In this case, analysts might never know about the existence of an encryption service. To prevent this kind of problem, security should be addressed in a phase where all the system components and functionalities are being modeled, i.e., in system architecture phase [Bass et al., 2003].

The most used paradigm to model the system architecture is the Object-Oriented (OO) approach, allowing the decomposition of complex systems in modular and functional components found in the system domain. However, the OO paradigm has some deficiencies when modeling concerns (functionalities) that interact with different system components. These are called crosscutting concerns. Security is a classical example of this kind of functionality.

One of the OO's extensions that resolve the crosscutting concern problem is the Aspect-Oriented approach. In this technique, behavior that affects the different classes is encapsulated in modular units, named Aspects, that improves clarity and understanding of crosscutting concerns. We show this approach in the next Section.

In this paper we present two patterns that will help to construct secure applications based in the aspect-oriented paradigm, to encapsulate the security and improve its legibility. The first pattern presented is responsible for access control to system functionalities. The second pattern presented is responsible for applying security in each system's functionality. Finally, we show how the interaction of these two patterns present solutions for the majority of security problems defined in other researches [Scott, 2004, Redwine and Davis, 2004, Win et al., 2001].

These patterns had been implemented and applied to an Online Accounting System (CETAV). In this system, small and medium size companies register purchases and sells, and they can obtain prediction reports about future sells. We will use this system to illustrate the pattern examples.

2. Aspect Orientation

Aspect-Oriented approach [Elrad et al., 2001] is a recent technique that resolve crosscutting concerns. Each concern is encapsulated in a modular unit, called *Aspect*, which has associated behaviors that affect multiple classes. With that, aspects improve modularization, present high cohesion and have a behavior similar to that of a class in the Object Oriented approach. An aspect also has attributes, methods, etc.

2.1. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP), allows for easier implementation code than Object-Oriented Programming (OOP) when dealing with concerns that are traversal to application code. It is important to point that AOP complements the OOP and in any way replace it. In AOP there are four basic and essential components [Kiczales et al., 2001]:

- **Join-Point:** Well-defined points in a program execution where a behavior will be added. This points might be constructor class, methods, exceptions, etc.
- **Pointcut:** Group a set of join points based on logics operator criteria.
- **Advice:** Responsible to add the behavior that will be executed when a point is intercepted in the program. An advice can capture one or more Pointcuts.
- **Aspect:** Contain the three components mentioned above. It has a behavior similar to that of a class and it is responsible to encapsulate the crosscutting functionalities that otherwise would be spread across the system.

It is important to notice that Advice component can intercept code in three different moments:*before* is executed before the method call, *around* is like *before* but it allows to cancel the method execution and, *after* is executed when finalized the method call.

Inside the advice, depending of the AOP languages, it is possible to have access to parameters of the intercepted method and to attributes or other methods of the class that contains the intercepted method.

2.2. Aspect-Oriented Modeling

Aspect-Oriented Modeling (AOM) provides the way to describe and communicate the specification of crosscutting concerns at analysis and design level. We use in our diagrams an extension to UML that avoid to change its metamodel and support AOM only using UML standard extension mechanism [Groher and Schulze, 2003]. In this extension, the notation includes three essential packages: *base package* containing the application domain classes intercepted by the aspects; *connector package* that encapsulates the underlying core concepts implemented in the technology to be used (like AspectJ or AspectC++), that means, this package is responsible to intercept the points where will be

implemented an behavior and; *aspect package* containing the crosscutting concern and the implementation of the behavior associated to the intercepted points. Like Figure 1 shows, base and aspect package have not direct connection and the link between them is given only by the connector.

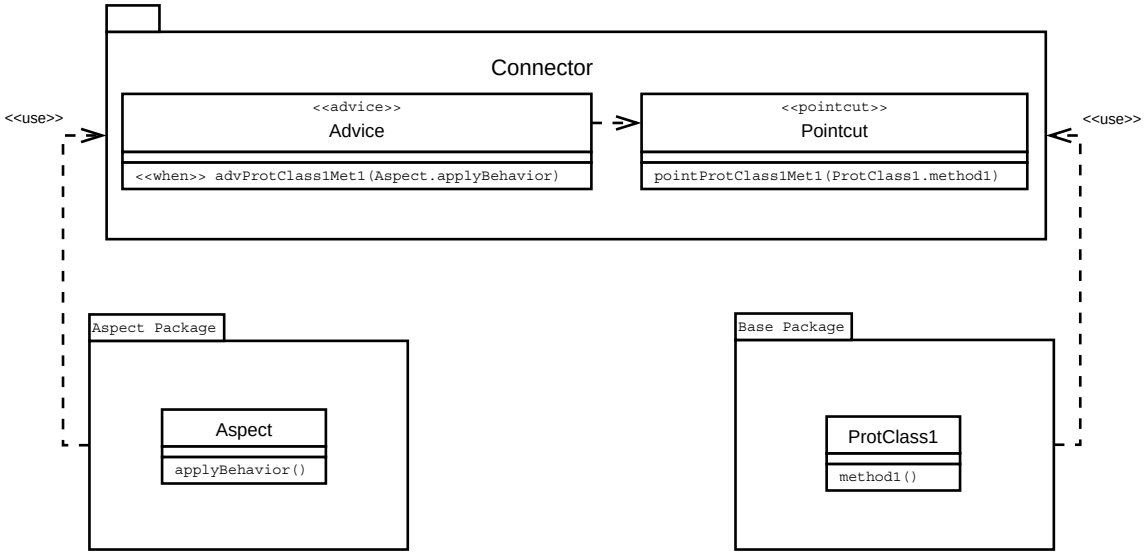


Figure 1: Aspect Oriented Modeling Example.

3. Patterns

In this section we describe two Aspect-Oriented architectural patterns that model security access control to application functionalities and put on the security issues on the architectural application level. This patterns are presented in POSA[1] format [Buschmann et al., 1996] and the implementation code use AspectJ [Kiczales et al., 2001] for the sake of simplicity.

3.1. Access Policy Control Pattern

The Access Policy Pattern provides a mechanism to abstract and encapsulate, in a modular unit, the protection against inappropriate access to system functionalities, involving in this process the organization's policies.

3.1.1. Example

Some CETAV functionalities should be only accessed by certain groups of users according to an organizational policy. For instance, electronic documents can be delivered only by users registered with the Accountant Role in the system. Therefore, other users should not be able to use this functionality.

In software development phases, these kind of situations, generally involve system functionality accesses, are very difficult to model and to implement. This is because the functionality is spread among various classes. For example, Figure 2 shows that the CETAV sell functionality is spread in four classes (shaded ellipses) that include GUI, Application Domain and Database layers. If all these four classes must control the access in order to

[1]Pattern-Oriented Software Architecture.

allow or deny the execution of a functionality, this control will be spread in these classes with consequent problems including tracking, difficult classes reutilization, high coupling between domain and security classes, and others.

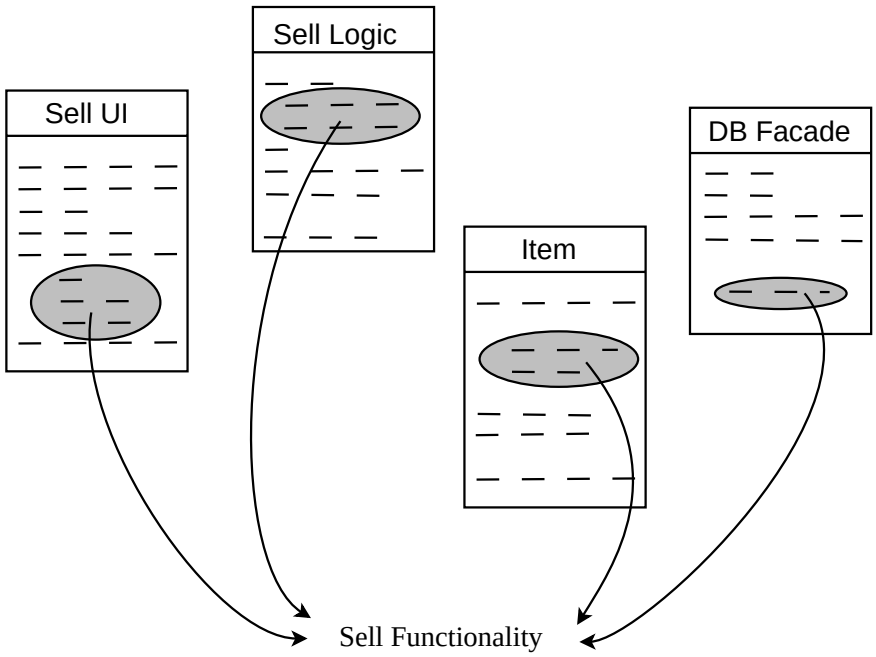


Figure 2: CETAV Sell Functionality spread among various classes.

3.1.2. Context

The Access Policy Control Pattern can be used whenever an application embeds (or requires to embed) security code to restrict access to system functionalities. This restriction can be handled in any way, but the main idea is to protect against the inappropriate use of system functionalities of inadequate users. It is important to highlight that this pattern focuses on the application level of an architectural view of the system.

3.1.3. Problem

Imagine you are designing a system that needs to perform access validations in different parts of the program. This is very common in systems with various users with different roles and permissions. These users want to access some functionalities but these functionalities are restricted for certain roles through the access policies established by the organization.

The design of the system, specially the access policy control, has to consider the following *forces*:

- The validation needs to be performed in a specific place and in a unified way, allowing maintainability and reusability of security code.
- Such systems need to establish an access control to system functionalities based on its organizational policies rules.
- The addition, modification and removal of access policies should be easily done because a system is related with organizational policies which can vary in time.
- How to keep track security validations through the system, allowing to identify security access holes.

3.1.4. Solution

Functionalities are accessed by method invocations. The solution is based on intercepting method invocations of the functionalities to be secured, using the principles of aspect oriented programming explained in Section 2.1.

The solution requires that each invocation to methods that need protection is intercepted. All these interceptions will be validated, against the organizational policies, in an aspect. The access validation code is abstracted from the application code by putting it into an external class (**Policy** Figure 3) that match the method intercepted with the policy for this method and returns to the caller if it is allowed or denied its access.

3.1.5. Structure

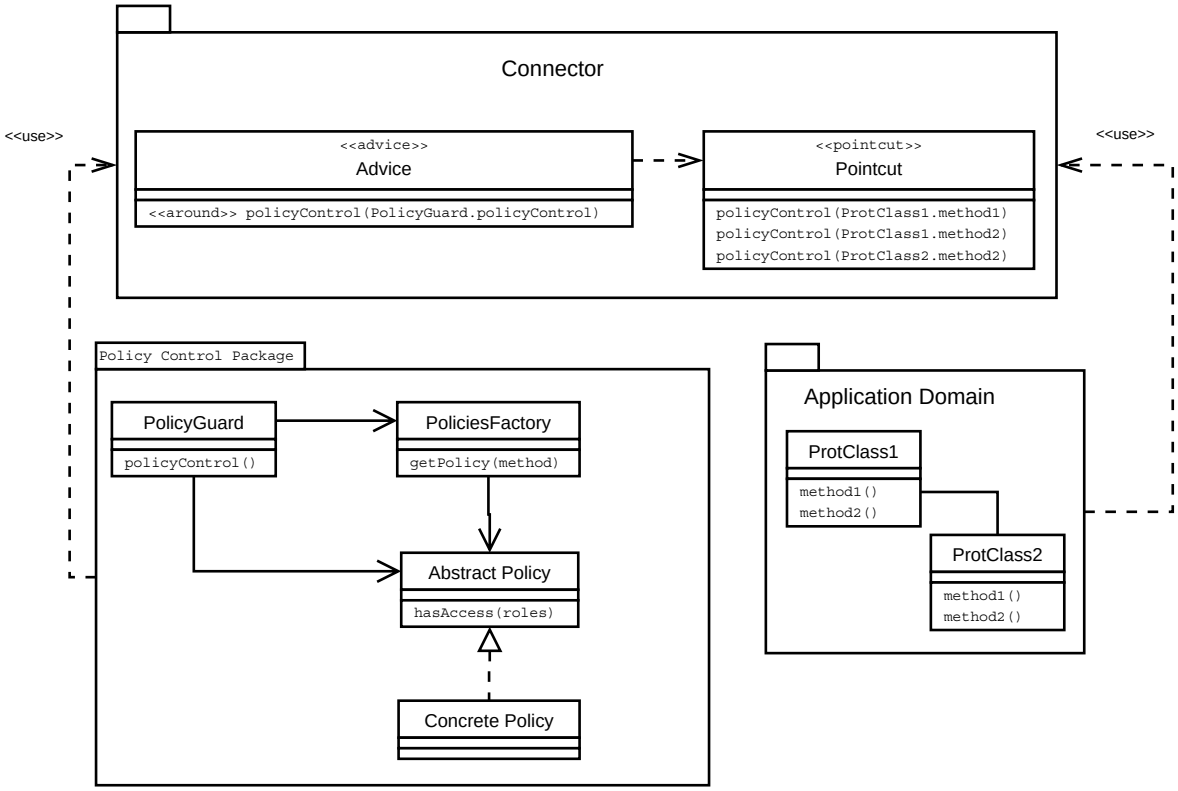


Figure 3: Access Policy Control Pattern.

Participants

We describe the participants involved in the structure of Figure 3.

Protected classes: Represent any classes in the application domain that require a control when accessing its methods.

Policy: Validates whether the access to a functionality must be allowed or denied. We used the Strategy Pattern[2] because this validation can be implemented in different ways.

[2]Strategy defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [Gamma et al., 1994].

PoliciesFactory: Given a method, it returns the **Policy** object for it. We used the Factory Method Pattern[3] to create the **Policy** objects.

PolicyGuard: This aspect intercepts method invocations at **Protected** classes. Interception code is executed before method code, and it is able to cancel method execution whenever the access is denied.

3.1.6. Implementation

```
1   Aspect PolicyGuard {
2
3       pointcut policyControl() : ProtClass1.method1() ||
4                                  ProtClass1.method2() ||
5                                  ProtClass2.method2() ;
6
7       around() : policyControl() {
8           Collection roles = Session.getCurrentUserRoles();
9           String method = getMethodID();
10          Policy policy = PoliciesFactory.getPolicy(method);
11          boolean allowed = policy.hasAccess(roles);
12          if(allowed) {
13              proceed();
14          } else {
15          //Inform the user that is access attempt was denied
16          Session.registerAccessDenied(Session.getCurrentUser(),
17                                       method);
18      }
19  }
```

Figure 4: Algorithm for the PolicyGuard aspect.

Figure 4 shows the intercepted behavior to be executed before the method. Methods being intercepted are listed in the *pointcut* construction (Lines 3-4). In lines 7-9, the user roles and the policy associated to the method are obtained, *getMethodID* function returns a string that uniquely identifies the intercepted method. This string is recognized by **PoliciesFactory** which returns a Policy object capable of verifying permissions. The **Policy** validates if any of the user roles contains the minimum permission required to access the method (Line 10). If it is allowed, *proceed* method lets the system continues its normal flow (Line 12). Otherwise, the user must be informed that his access attempt was denied. Finally, in this code, any procedure to deal with access denied can be included(Line 15).

Like we describe in Section 3.1.5, the Policy class was implemented using the Strategy Pattern. There are some strategies that could be used to validate the access to a certain functionality, for example: *Access Matrix Control* (ACM) [Harrison et al., 1976] and *Basic Role Control* (BRC). The former use a matrix users versus resources that define the rights allowed to a certain system user. The latter, specifies a set of *basic roles*[4] with access to each functionality. Given a role, it is verified whether one of its sub-role is one of the specified basic roles for the policy. The basic role found will be used by the system instead of the original user role, ensuring the principle of least privilege proposed by [Saltzer and Schroeder, 1975].

[3]Factory Method defines an interface for creating an object, but let subclasses decide which class to instantiate [Gamma et al., 1994].

[4]Given a Role hierarchy, a basic role is a role with no sub-roles [Sandhu et al., 1996].

3.1.7. Example Resolved

CETAV was originally an OO system without security mechanisms for functionality access. It is important to notice that this aspect-oriented implementation, is very useful when having a system lacking of control in functionality access. It introduces minimal changes in application code and in original system architecture, because it just adds an independent component outside this original architecture.

In order to validate the access to functionalities we used the **PolicyGuard** aspect, based on the Protected System Pattern [Halkidis et al., 2004]. This pattern requires a component to manage users and their roles. Such component could be implemented using the RBAC model [Sandhu et al., 1996] and hold in an instance of the Session Pattern [Yoder and Barcalow, 1997] making this information available to the **PolicyGuard**.

Figure 5 shows the class diagram of the solution. When *addNewSell(Sell)* or *removeSell(Sell)* methods of the **SellModule** class are called by the CETAV system, a **PolicyGuard** aspect intercepts them in order to allow or deny the access, so users must pass through this guard to access these functionalities. The interception point is declared in the **Pointcut** and the steps that follow the interception was explained in the algorithm presented in Section 3.1.6.

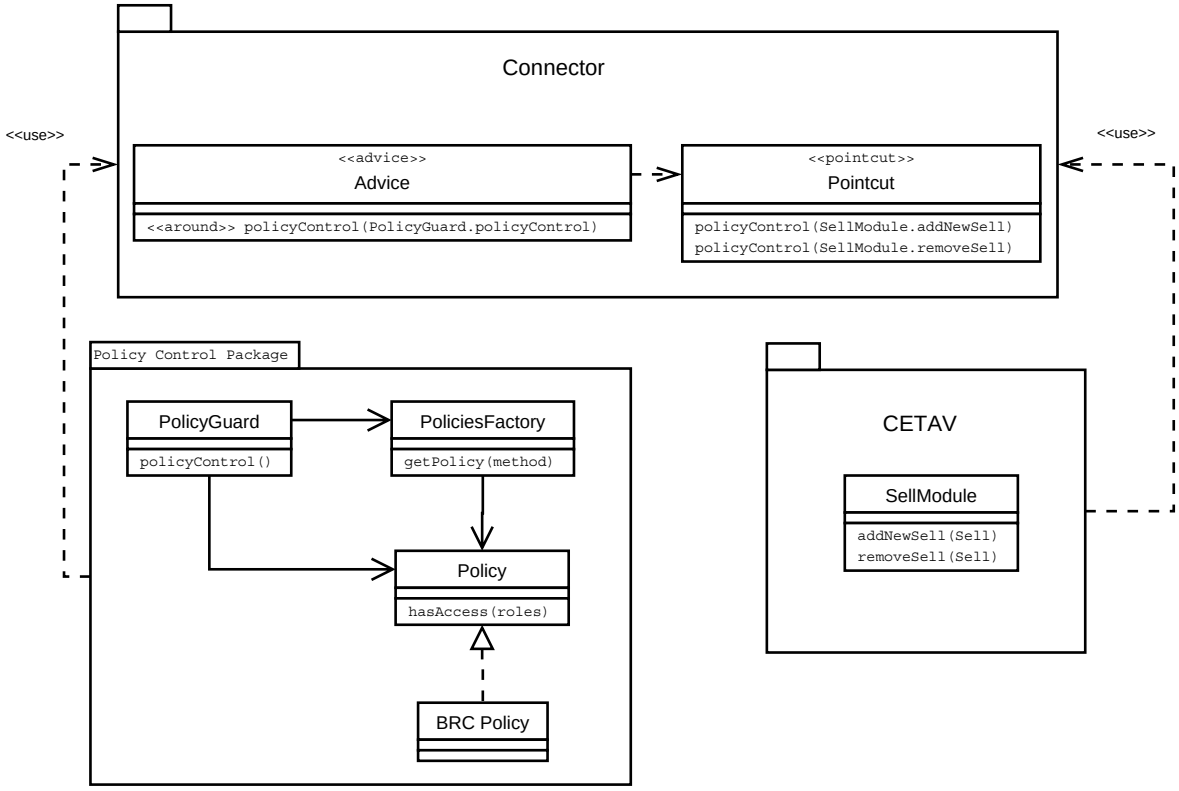


Figure 5: Access Policy Control applied in CETAV.

3.1.8. Known Uses

We implemented the Access Policy Control Pattern in CETAV system. CETAV's policy control was originally implemented with an object-oriented approach using security patterns [Yoder and Barcalow, 1997]. After applying this aspect-oriented pattern, we measured cohesion and coupling using metrics proposed by [Tsang et al., 2004]. The aspect-oriented implementation of access policy control shows being more cohesive than the

object-oriented one. Domain classes' coupling was reduced, because they no longer include policy control code, since it was extracted to **PolicyGuard**.

The Ariel Project [Pandey, 2005] presents an alternative to enforce security validation through a declarative policy language to specify a set of constraints on accesses to resources [Pandey and Hashii, 1999]. It provides fine-grained access control for mobile java programs applying a set of code transformation tools enforcing these constraints directly on the code, similar to aspect-oriented approach.

In [Westphall and Fraga, 1999] the authors presented an authorization scheme for large scale networks that involves programming models and tools represented by Web, Java and CORBA for security [Object Managment Group, 2002]. The access validation is provided by having CORBA interceptor classes that capture method calls applying authorization defined by control policies. These ideas were implemented in JaCoWeb [Fraga, 2005].

3.1.9. Consequences

The Access Policy Control Pattern has some important **benefits**:

- Access validation to system functionalities is encapsulated by the Access Policy Control Pattern in the PolicyGuard Aspect, so this validation is not spread among the application domain classes as in OO approach is usually done. Alternatives, like [Fernandez et al., 2005], also separate this validation mapping each use case, that needs to enforce access, to a new class that control this validation. This approach introduces some complexity in the design model and consequently in the implementation stage.
- Legacy systems can have this pattern applied to establish an appropriate security control. Policy class perform the validation of an access, based in the organizational policies.
- Given the policy encapsulation offered by the pattern, incorporation of new policies or modification of existing ones has no direct impact in system architecture. Only will be necessary to add (or modify) a policy in the Policy class and add the interception point in the PolicyGuard Aspect.
- Interception points of the methods whose access is controlled will be centralized in the Access Policy Control Pattern. This helps to identify security holes, and to keep track of security validations through the system.

The **liabilities** of this pattern are as follows:

- Our approach is more complex because it controls security across applications. The authorization rules must be applied to all applications that access some shared data.
- Aspect-Orientation, nowadays, is less used than Object-Orientation. In some cases our approach could not be applied because Aspect-Orientation techniques are not available.
- A general Aspect-Orientation drawback is that the code inside the aspect is very coupled with the intercepted code minimizing the reusability of the aspect (**PolicyGuard**). In the other hand the security specific code (**Policyfactory** and **Policy** classes) can be reused due to it is implemented outside the aspect.

3.1.10. See also

Access Policy Control includes features of Protected System and Policy Pattern, described in [Blakley and Heath, 2004], as well as Checkpoint and Roles Pattern proposed by [Yoder and Barcalow, 1997].

Access Policy Control Pattern uses the Session Pattern [Yoder and Barcalow, 1997] to store and get user roles.

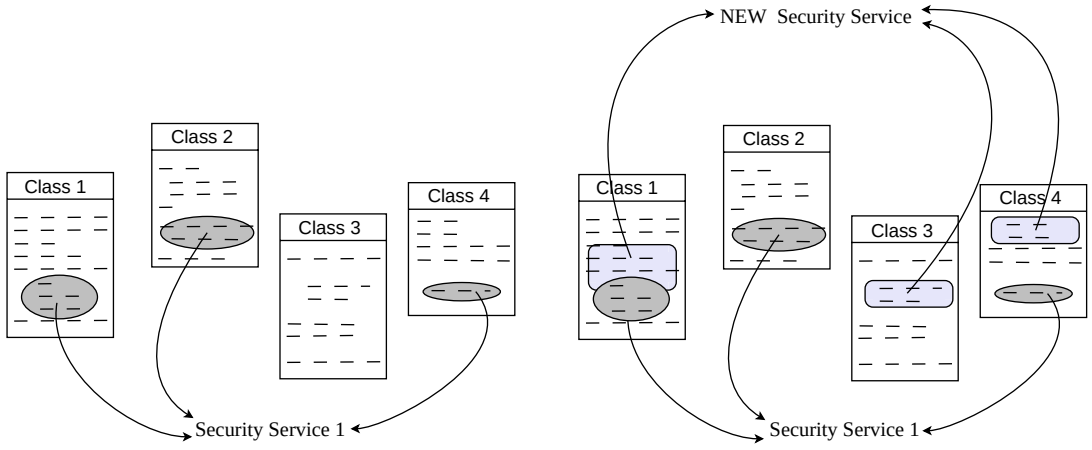
A Policy Guard is an aspect-oriented instance of **Reference Monitor** [Fernandez and Pan, 2001].

3.2. Security Services Pattern

The Security Services Pattern provides a mechanism to protect system resources through a set of security services. Each one of these services is abstracted and encapsulated in a modular way providing the means to clarify what resources are protected by what services.

3.2.1. Example

The CETAV system, in order to send a document, requires document encryption and registration of the action for a future audit. Suppose that this system was constructed with an OO approach and with two security services (authenticate, audit) spread throughout the code. Like software complexity increased, minimal changes to these services produced very high costs, in time, to discover and change the classes involved (shaded ellipses in Figure 6(a)). Another problem arise when new service is add (encrypt the document), the new code must ensure that this new service does not affect the old security functionalities and will have to spread this new security services throughout the classes (shaded boxes in Figure 6(b)).



(a) Tracking domain classes involving a certain security service

(b) Add a new security service involving various domain classes

Figure 6: Problems raised in CETAV Object-Oriented approach.

3.2.2. Context

The Security Services Pattern is useful when it is necessary to apply a set of related security services on a application level functionalities. Each one of these services are

encapsulated in a modular way.

3.2.3. Problem

Imagine you are modeling a system where you need to apply security services to methods that need to protect its resource confidentiality and integrity. For example, suppose that a system needs to protect some data and this protection is to audit the actions upon the resource, encrypt it and communicate through a secure communication channel when interact with another system. We can expect that some methods only audit the data, others only encrypt the data and others do both.

Now, suppose that you need to add in such systems a new security services, like validate that a sent resource doesn't have any sensible data (Input/Output validation) which might open security holes in the system.

We can see, from an architectural and design point of view, that this protection involves a set of related security services and it is difficult to track which security services were applied to what methods. This happen because the security is spread throughout the system functionality.

The design of the system, specially the security services, has to consider the following *forces*:

- The security service applied in a system functionality should be centralized in order to improve the reusability of the application domain code and the security service service.
- The addition and modification of a security service should be easy because new security holes can be discovered very frequently.
- Easily keep track which system functionalities are secured by each security service.

3.2.4. Solution

All the functionalities that need a certain security service are intercepted by method invocations using the principles of aspect oriented programming explained in Section 2.1.

The solution requires that each security service will be in a module that control the interceptions and can apply the behavior associated to it. With this approach, we ensure that the service implemented is highly cohesive because all the actions to that service are in one module and totally modular because there are not relationship with others modules.

3.2.5. Structure

Participants

We describe the participants involved in the structure of Figure 7.

Protected Class Represent any clas in the application domain that requires a security service over their methods.

Security Package Consists of a general implementation of security concerns. It might be an external library or an application subsystem.

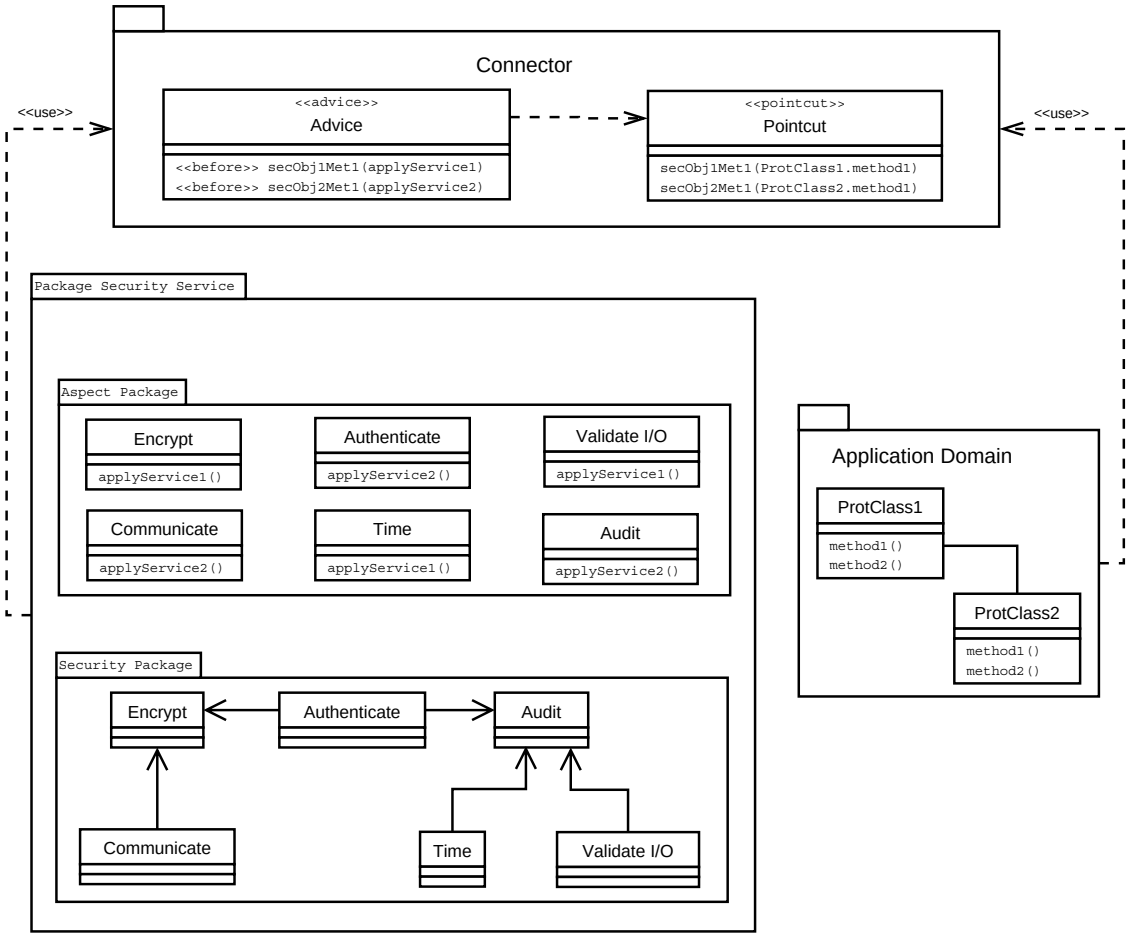


Figure 7: Security Services Pattern.

SecurityService (Encrypt, Communicate, etc.) Applies a particular security service in a functionality of the system. Any of these aspects act as the Protected System Pattern.

Aspects Package It is defined by a set of **SecurityService** aspects. It represents the way the security is included in the application domain.

3.2.6. Implementation

Figure 8 shows the implementation of two security services in an existing module of CETAV. The original code was not altered, the security services were implemented in the aspect code that intercepts the original methods.

The aspects **Audit** and **Encrypt** apply respectively auditing (Line 1) and encryption (Line 11) services to the send() method of the **SendModule** class in Figure 9. The send method is intercepted in the *pointcut* construction (Lines 2 and 12). The Audit service code (*secure_send* method), which is called before the *send* method. It logs the send action, storing the destiny, the file name and the date and time when this action occurred (Lines 5 to 7). The **Encrypt** service encrypts the file with an external library (Line 17).

As it can be seen in this example, the original code is not modified and the security specific code can remain encapsulated in specialized classes. The relationship between the application domain and the security services is done inside the aspect and is the only highly coupled code.

```
1   public Aspect Audit {
2       pointcut secure_send() : SendModule.send(destiny, file);
3
4       before() : secure_send() {
5           InetAddress destiny = getParameter("destiny");
6           File file = getParameter("file");
7           logSend(destiny, file, session.getDate("today"));
8       }
9   }
10
11  public Aspect Encrypt {
12      pointcut secure_send() : SendModule.send(destiny, file);
13
14      before() : secure_send() {
15          File file = getParameter("file");
16          Key theKey = Authenticate.getKey(session.getUserKey());
17          file = Encrypt.encrypt(file, theKey);
18      }
19  }
```

Figure 8: Applying Security Services in CETAV.

3.2.7. Example Resolved

This pattern, applied to CETAV, was very useful to trace the functionalities that a service intercept and to separate the security code from the business logic code. Also, it allowed us to add a new service in a easy way. In order to resolve the problems presented in the example, the aspects **Encrypt** and **Audit** can be used to intercept method invocations that need to apply these security services.

Figure 9 shows the class diagram of the solution. The **SendModule** class contains the functionality to be secured. When *send* method defined in the **SendModule** class is called by the CETAV system, the **Encrypt** and the **Audit** aspects intercept it. The **Encrypt** aspect encrypts the file passed as a parameter using the private key generated by the class **Authenticate**. This encryption is provided by an **Encrypt** class of an external library. The **Audit** aspect registers this sending, storing the destiny and the date in a log. The interception point is declared in the **Pointcut** and the steps that follow the interception was explained in the algorithm presented in Section 3.2.6.

3.2.8. Known Uses

This pattern is often implemented as a set of security services, where each one might use a pattern to deal with its respective service.

We implemented the Security Services Pattern in CETAV system. The encryption and authentication services were originally implemented by CETAV with an object-oriented approach. Our aspect-oriented implementation extracted the spread security code, reducing the system domain coupling and incrementing its cohesion.

The Lumbago [Koshiba, 2001] system is an application used to maintain information records about patients in healthcare institutions. The information maintained by the system is highly confidential so it needs to be secured, and it applied encryption mechanisms to secure it. The original application was built with no encryption in mind. The method used was to create an aspect that abstracts the security code taking care of this service, minimizing the impact on the original implementation.

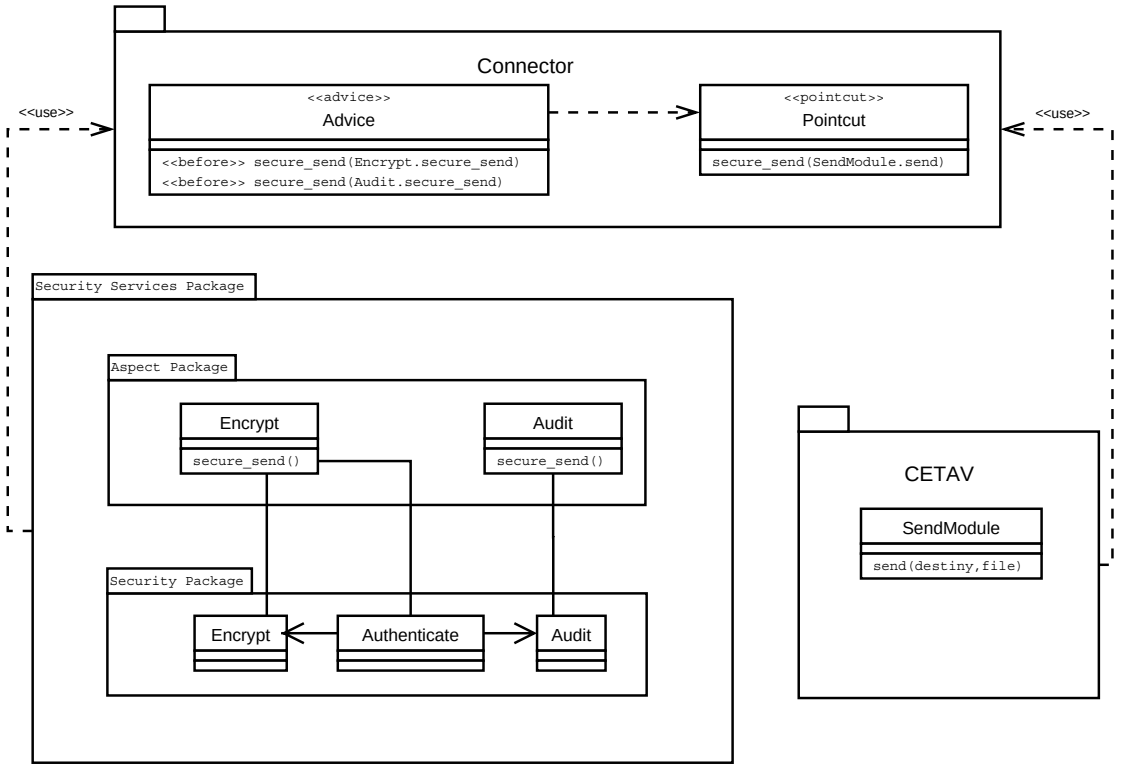


Figure 9: Security Services applied in CETAV.

3.2.9. Consequences

The Security Services Pattern has some important **benefits**:

- It is easy to identify which functionalities of the system have a certain type of security service. For each service there is an aspect that take care and encapsulate the resource protection.
- New types of security services can be added easily, as it was described in the implementation example. Each security service is independently implemented of the others, because each one is encapsulated in its own aspect. To provide a new security service we only need to model a new aspect that provide the behavior necessary to protect the resource, to make it capture the code being secured and, to implement the security specific code.

The **liabilities** of this pattern are as follows:

- It is difficult to track which types of security services were applied to a system functionality. This is due to the separation of the services into different aspects.
- If the system already contains some implemented security mechanism, the insertion of this pattern might be difficult since it will be necessary to extract all the security code, spread among the domain classes, and encapsulate them in their respective aspects. However, applying this pattern will be relatively simple if the system does not contain a security mechanism.

3.2.10. See also

Communicate aspect uses the Secure Access Layer [Yoder and Barcalow, 1997] and the Secure Communication Pattern [Blakley and Heath, 2004, Braga et al., 1998] in order to communicate with third parties in a secure channel.

Authenticate aspect uses the Session Pattern [Yoder and Barcalow, 1997] to store data information related to authentication, like keys or credentials.

Validate I/O aspect uses the Object Filter Pattern [Hays et al., 2000] to filter undesirable data.

Encrypt aspect uses the Information Secrecy Pattern and Message Integrity [Braga et al., 1998] in order to keep the data integrity and secrecy.

Each security service aspect acts as a Protected System Pattern [Halkidis et al., 2004].

4. Integrating the Patterns as an Architecture

The architecture represents a system from a global point of view and defines the general features that will be used in the different phases of the development process. These features are defined in an abstract level, and it makes possible to understand the system being modeled. Being the security an important issue, it is recommendable to contemplate it in this model [Fernandez et al., 2005].

For this reason, we integrated the Access Policy Control Pattern and the Security Services Pattern to show how they can be integrated in the architectural model of a system. This integration is needed, because as shown in other researches [Gao et al., 2004, McGraw and Viega, 2002] many security problems arise when it is not dealt as a part of a system since initial development phases.

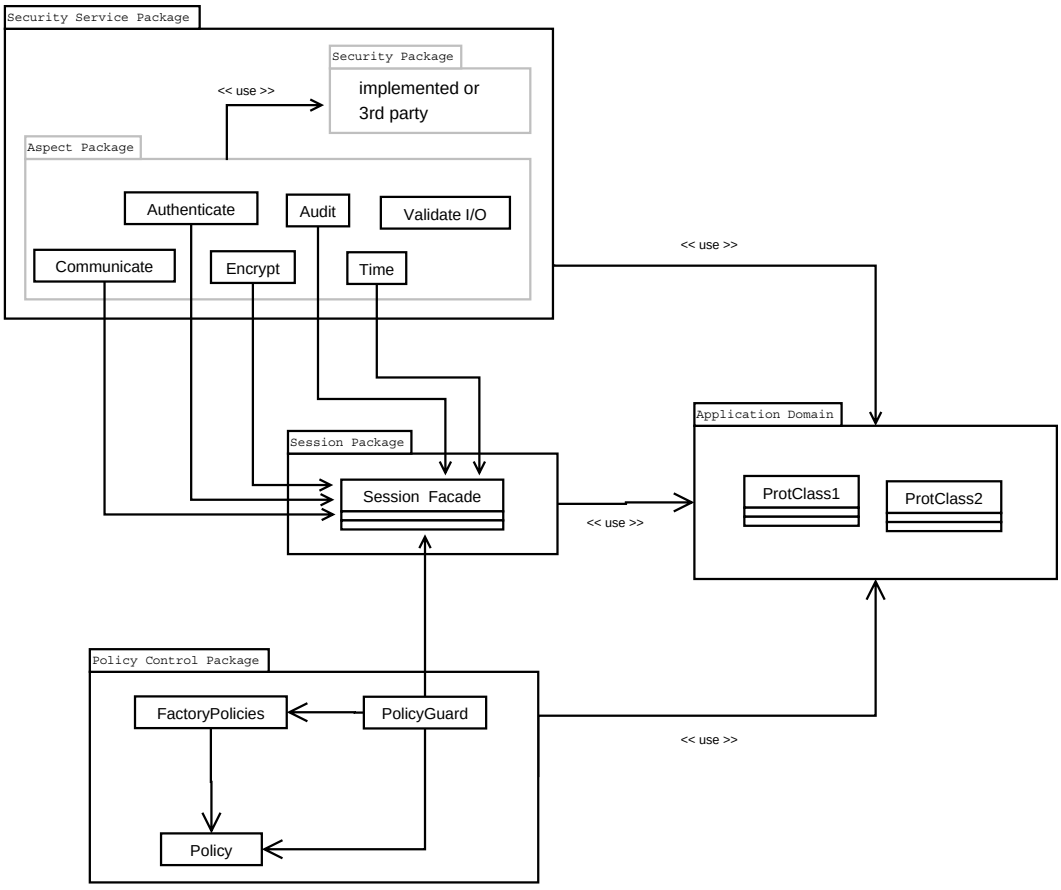


Figure 10: Architecture.

Figure 10 presents the Access Policy Control Pattern and instances of the Security Services Pattern. They do not interact directly because methods intercepted by the former need not necessarily be intercepted by the latter. To interact with the application both

patterns can use an instance of the Session pattern [Yoder and Barcalow, 1997]. The Session pattern Facade is used to encapsulate user information and some system configuration values, such as current session time or user roles. It is important that the aspects have a centralized point of access to the user and system information to try to reduce the coupling of the aspect code with the application.

A clear advantage of this architecture is that the implementation of security issues is separated of the application domain, being the aspects the connection point between them. If security is not defined in an architectural model it becomes difficult to get these benefits [Fernandez et al., 2005].

5. Conclusions

Security is a crosscutting concern problem. This motivated us to use the aspect-oriented technique that deals with this kind of problem better than the object-oriented approach. Based in this approach, we proposed two architectural patterns: the Access Policy Control pattern and the Security Services pattern.

The Access Policy Control pattern ensure that all functionalities accesses are controlled by a set of role-based organizational policies. This security control is encapsulated in a modular unit improving the security code reusability.

The Security Services pattern allows to apply a set of security services, like encryption and authentication, to any functionality of the system that requires it. Each of these security services are centralized in a modular unit, improving the application and security code reusability.

Finally, we interrelate these two patterns in order to cover the principal security problems found in many systems, that normally arise when security is not considered as a part of a system in the architecture model.

Acknowledgments

We would like to thank our colleagues at University of São Paulo for their comments and suggestions, specially to Eduardo Guerra who provided valuable guidance at the architecture specification. Eric Ross helped us implementing the Patterns in CETAV. We would also like to mention the pertinent and focused suggestions given by Eduardo B. Fernandez, our shepherd in the final phase of PLoP submission.

References

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley Professional, second edition.

[Blakley and Heath, 2004] Blakley, B. and Heath, C. (2004). *Security Design Patterns*. The Open Group.

[Braga et al., 1998] Braga, A., Rubira, C., and Dahab, R. (1998). Tropyc: A pattern language for cryptographic software. In *Proceedings of the 5th Conference on Patterns Language of Programming (PLoP´98)*.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England.

[Elrad et al., 2001] Elrad, T., Filman, R., and Bader, A. (2001). Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32.

[Fernandez and Pan, 2001] Fernandez, E. and Pan, R. (2001). A pattern language for security models. In *Proceedings of the 8th Conference on Pattern Languages of Programming (PLoP 01)*.

[Fernandez et al., 2005] Fernandez, E., Sorgente, T., and Larrondo-Petrie, M. (2005). A uml-based methodology for secure systems: The design stage. In *Third International Workshop on Security in Information Systems (WOSIS)*, Miami.

[Fraga, 2005] Fraga, J. D. S. (2005). JaCoWeb project. `http://www.lcmi.ufsc.br/jacoweb/`.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, R. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Adisson Wesley.

[Gao et al., 2004] Gao, S., Deng, Y., Yu, H., He, X., Beznosov, K., and Cooper, K. (2004). Applying aspect-orientation in designing security systems: A case study. In *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, Canada.

[Groher and Schulze, 2003] Groher, I. and Schulze, S. (2003). Generating aspect code from UML models. In *The 4th AOSD Modeling With UML Workshop*.

[Halkidis et al., 2004] Halkidis, S., Chatzigeorgiou, A., and Stephanides, G. (2004). A qualitative evaluation of security patterns. In *Proceedings of the 6th International Conference, ICICS 2004*, Malaga, Spain.

[Harrison et al., 1976] Harrison, M., Ruzzo, W., and Ullman, J. (1976). Protection in operating systems. *Commun. ACM*, 19(8):461–471.

[Hays et al., 2000] Hays, V., Loutrel, M., and Fernandez, E. (2000). The object filter and access control framework. In *Proceedings of the 7th Conference on Patterns Language of Programming (PLoP´00)*.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). Getting started with AspectJ. *Commun. ACM*, 44(10):59–65.

[Koshiba, 2001] Koshiba, T. (2001). A new aspect for security notions: Secure randomness in public-key encryption schemes. In *PKC '01: Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography*, pages 87–103, London, UK. Springer-Verlag.

[McGraw and Viega, 2002] McGraw, G. and Viega, J. (2002). *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison Wesley.

[Object Managment Group, 2002] Object Managment Group (2002). Security service specification. Technical report, Object Managment Group. (version 1.8).

[Pandey, 2005] Pandey, R. (2005). The ariel project. `http://pdclab.cs.ucdavis.edu/projects/ariel/`.

[Pandey and Hashii, 1999] Pandey, R. and Hashii, B. (1999). Providing fine-grained access control for java programs. In Springer-Verlag, editor, *Proceedings of the 13th Conference on Object-Oriented Programming ECOOP'99*, Lecture Notes in Computer Science, Lisboa, Portugal.

[Redwine and Davis, 2004] Redwine, S. and Davis, N. (2004). Processes to produce secure software. Technical report, National Cybersecurity Partnership Task Force Report.

[Saltzer and Schroeder, 1975] Saltzer, J. and Schroeder, M. (1975). The protection of information in computer systems. *Proceedings of the IEEE,* 63(9):1278–1308.

[Sandhu et al., 1996] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. (1996). Role-based access control models. *IEEE Computer,* 29(2):38–47.

[Scott, 2004] Scott, D. (2004). *Abstracting Application-Level Security Policy for Ubiquitous Computing*. PhD thesis, University of Cambridge.

[Tsang et al., 2004] Tsang, S., Clarke, S., and Baniassad, E. (2004). An evaluation of aspect-oriented programming for java-based real-time systems development. In *Proceedings of the 7th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Austria.

[Westphall and Fraga, 1999] Westphall, C. and Fraga, J. (1999). Authorization Schemes for Large-Scale Systems based on Java, CORBA and Web Security Models. In *The IEEE International Conference on Networks*, pages 327–334, Brisbane-Queensland, Australia.

[Win et al., 2001] Win, B. D., Vanhaute, B., and Decker, B. D. (2001). Security through aspect-oriented programming. In *Network Security*, pages 125–138.

[Yoder and Barcalow, 1997] Yoder, J. and Barcalow, J. (1997). Architectural patterns for enabling application security. In *Proceedings of the 4th Conference on Patterns Language of Programming (PLoP´97)*, volume 2.

Propagação Direcional para Processamento de Imagens

Francisco de Assis Zampirolli[1] **, Roberto de Alencar Lotufo**[2] **,**
Lucas Padovani Trias[1]

[1]Centro Universitário Senac
Av. Eng. Eusébio Stevaux, 823 – 04696–000 – São Paulo, SP

[2]FEEC – Faculdade de Elétrica e de Computação – UNICAMP
6101 – 13083–970 Campinas, SP

{francisco.zampirolli,lucas.trias}@sp.senac.br, lotufo@unicamp.br

Abstract. *This paper describes an extension of a pattern for image processing, where we consider a propagation direction. This extension may be applied on many morphological operators like Euclidian Distance Transformation (EDT). In this work we present simple and efficient ways to implement EDT using the directional propagation of erosion and decomposition the structuring function in* 3×3*.*

Resumo. *Este artigo descreve uma extensão de um padrão por propagação para processamento de imagens, onde consideramos uma direção de propagação. Esta extensão pode ser usada em vários operadores morfológicos, como dilatação, erosão e Transformada de Distância Euclidiana (TDE). Neste trabalho apresentamos formas simples e eficientes de implementar a TDE usando a erosão por propagação direcional e usando a decomposição da função estruturante em* 3×3*.*

Palavras-Chave: morfologia matemática, desenvolvimento de software, técnicas de implementação, padrões de algoritmo e programação genérica.

Introdução

Neste artigo falaremos sobre as formas de realizar operações frequentes na área de processamento de imagens. Por se tratar de uma área muito específica, a seguir formalizaremos alguns conceitos e definições que serão utilizados durante a descrição do padrão.

a) Morfologia Matemática

Uma forma elegante de resolver problemas de processamento de imagens é através da utilização de uma base teórica consistente. Uma destas teorias é a morfologia matemática criada na década de 60 por Jean Serra e George Matheron na *École Nationale Superiéure des Mines de Paris*, em Fontainebleau, França. Esta teoria diz que é possível fazer transformações entre reticulados completos[1], os quais são chamados de operadores morfológicos. Na morfologia matemática existem quatro classes básicas de operadores, chamados de operadores elementares: dilatação, erosão, anti-dilatação e anti-erosão.

Iremos aplicar a propagação direcional no operador erosão. Estudo análogo pode ser feito nos demais operadores elementares.

[1]Um conjunto qualquer com uma relação de ordem é um reticulado completo se todo subconjunto não vazio tem um supremo e um ínfimo. Para detalhes da teoria dos reticulados veja [Birkhoff, 1967].

b) Erosão Morfológica

A erosão atribui o menor valor de uma região predefinida (elemento estruturante) ao pixel que está sendo erodido. Deste modo o fundo da imagem consome as bordas dos objetos erodindo-os. Matematicamente ela é definida da seguinte forma:

Seja $\mathbf{Z}$ o conjunto dos inteiros, $\mathbf{E} \subset \mathbf{Z}^2$ o domínio da imagem e $K = [0, k] \subset \mathbf{Z}$ um intervalo de números inteiros representando os possíveis níveis de cinza da imagem. O operador erosão em níveis de cinza invariante por translação, $\varepsilon_b : K^{\mathbf{E}} \rightarrow K^{\mathbf{E}}$ ($K^{\mathbf{E}}$, é o conjunto de funções de $\mathbf{E}$ in K), é definido como [Heijmans, 1991]:

$\forall f \in K^{\mathbf{E}}, \forall x \in \mathbf{E}$ e $\forall b \in \mathbf{Z}^{\mathbf{B}}$,

$$\varepsilon_b(f)(x) = \min\{f(y)\dot{-}b(y-x) : y \in (B+x) \cap \mathbf{E}\}, \quad (1)$$

onde $B \subseteq \mathbf{E} \oplus \mathbf{E}$ é chamado *elemento estruturante*, o símbolo $\oplus$ é chamado soma de Minkowski, $B + x = \{y + x,\ y \in B\}$ (translação de B por x) e b é uma *função estruturante* definida em B por $b : B \rightarrow \mathbf{Z}$. Sejam v e t inteiros, definimos $t \rightarrow t\dot{-}v$ em K por

$$\begin{cases} t\dot{-}v = 0 & se\ t < k\ e\ t - v \leq 0; \\ t\dot{-}v = t - v & se\ t < k\ e\ 0 \leq t - v \leq k; \\ t\dot{-}v = k & se\ t < k\ e\ t - v > k; \\ t\dot{-}v = k & \forall v \in \mathbf{Z}. \end{cases}$$

Na Figura 2 mostramos um exemplo do efeito da erosão morfológica de uma imagem binária (com níveis de cinza preto e branco) usando a função estruturante apresentado na Figura 1.

0	0	0
0	0	0
0	0	0

Figura 1: Função estruturante b, com domínio 3 $\times$ 3

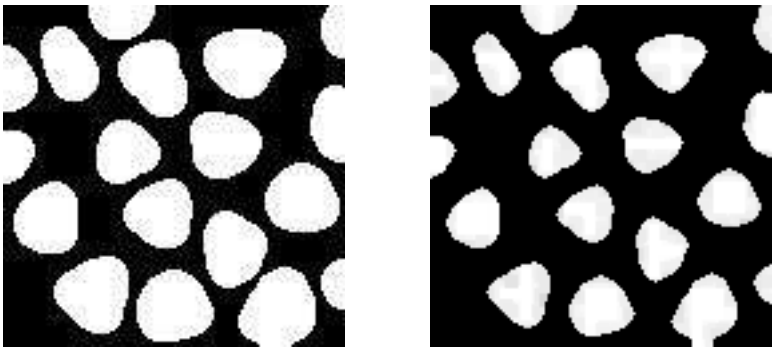


Imagem Original Imagem Erodida

Figura 2: Exemplo de erosão usando a função estruturante a Figura 1

c) Algoritmos que Implementam a Erosão Morfológica

A literatura fala sobre diversos modos de implementar a erosão morfológica.

O mais intuitivo e simples de ser implementado é o *algoritmo paralelo*. Nele a imagem (matriz) é varrida em qualquer ordem (inclusive, os pixels podem ser processados

de forma paralela), e para cada pixel a vizinhança é verificada aplicando-se a função estruturante. Esta implementação tem um desempenho baixo já que não utiliza nenhuma estratégia de otimização.

Buscando um melhor desempenho foram criados dois outros algoritmos o *sequencial* e o *por propagação*. No primeiro a imagem é percorrida sequencialmente. O ganho de desempenho reside no fato de a função estruturante ser dividida em dois e cada metade da função varrer metade da imagem. A metade superior da função estruturante varre a imagem descendo e a metade inferior subindo.

O *algoritmo por propagação* centra seu funcionamento em processar apenas os pixels que realmente fazem parte da borda que será erodida. Com isto não é desperdiçado tempo computacional tentando erodir pixels que não podem ser erodidos. Deste modo há uma redução significativa no número de pixels que serão verificados e consequentemente uma redução no tempo computacional gasto.

No *algoritmo por propagação* as coordenadas destes pixels são usualmente armazenadas em um conjunto[2] e são chamados de borda ou fronteira. A eficiência máxima deste tipo de algoritmo ocorre quando é possível que uma iteração de erosão gere a fronteira para a próxima iteração. Em cada iteração é aplicada a erosão tradicional a todos os pixels armazenados no conjunto. Quando este conjunto encontra-se vazio significa que não há mais fronteira a ser erodida, ou seja, toda a erosão possível para aquela iteração já foi feita.

Abaixo está o pseudocódigo que retorna a fronteria de propagação de uma imagem f usando a vizinhança definida pela função estruturante b. Esta borda é colocada no conjunto ∂f_b.

Function $\partial f_b = front(f, b)$
  for all $x \in \mathbf{E}$
    $\partial f_b = \{x : \exists y \in (B + x) \cap \mathbf{E},\ f(y) > f(x) \dot{-} b(x - y)\}$;

Abaixo está o pseudocódigo para a erosão por propagação, próprio para uso iterativo usando a mesma função estruturante b. Os parâmetros de entrada são a imagem f, a função estruturante b e a sua fronteira de f, ∂f_b. Os parâmetros de saída são a imagem g (erosão) e a sua fronteira ∂g_b.

Function $[g, \partial g_b] = eroPro(f, b, \partial f_b)$
  {g e ∂g_b são parâmetros de saída}
  $g = f$;
  for all $x \in \partial f_b$
    for all $y \in (B + x) \cap \mathbf{E}$
      if $g(y) > f(x) \dot{-} b(x - y)$
        $g(y) = f(x) \dot{-} b(x - y)$;
        if $y \notin \partial g_b$, $set\_in(\partial g_b, y)$;

onde $set\_in(\partial g_b, y)$ é a função para inserir y no conjunto ∂g_b. Observe que antes de inserir um ponto no conjunto de fronteira, é feito uma verificação para que o ponto não seja inserido mais de uma vez desnecessariamente. Veja um exemplo de uso deste algoritmo na Figura 3.

[2]Na literatura, o uso de operações por propagação é confuso pois em algumas vezes usa-se fila, em outras fila hierárquica. Para o caso da erosão, a fronteira é um conjunto que pode ser processado em qualquer ordem (análogo ao ocorrido no caso paralelo) e neste caso uma estrutura de conjunto (por exemplo, vetor) é suficiente para armazenar a fronteira.

$$\begin{matrix} k & k & k & k & k & k & k & k & k \\ k & k & k & k & k & k & k & k & k \\ k & k & k & k & k & k & k & k & k \\ k & k & k & k & [0] & k & k & k & k \\ k & k & k & k & k & k & k & k & k \\ k & k & k & k & k & k & k & k & k \\ k & k & k & k & k & k & k & k & k \end{matrix}$$

(a) f

$$\begin{matrix} k & k & k & k & k & k & k & k & k \\ k & k & k & k & k & k & k & k & k \\ k & k & k & k & [1] & k & k & k & k \\ k & k & k & [1] & 0 & [1] & k & k & k \\ k & k & k & k & [1] & k & k & k & k \\ k & k & k & k & k & k & k & k & k \\ k & k & k & k & k & k & k & k & k \end{matrix}$$

(a) $eroPro(f, b_4, \partial f_{b_4}) = \varepsilon^p_{b_4}(f)$

Figura 3: (a) Imagem de entrada f, onde o valor entre colchetes pertence à fronteira ∂f_{b_4}; (b) erosão por propagação por b_4, onde os pixels entre colchetes pertencem à nova fronteira $\partial(\varepsilon^p_{b_4}(f))_{b_4}$. b_4 é a métrica *city-block*.

É possível generalizar o algoritmo de erosão iterativa por propagação usando uma seqüência de erosões com funções estruturantes não crescentes. A condição para que esta generalização seja válida pode ser encontrada em Zampirolli [Zampirolli, 2003].

Contexto

Precisamos erodir imagens com eficiência. Existem técnicas de varredura da imagem que otimizam o processo de erosão, mas ainda assim eles não atingem a eficiência necessária.

Problema

Os algoritmos tradicionais que implementam a erosão morfológica desperdiçam tempo computacional processando pixels desnecessários. O *algoritmo paralelo* é extremamanete ineficiente por sua varredura e por sua forma de processar os pixels. O *algoritmo sequencial* é mais eficiente, mas seu grande ganho de desempenho restringe seu uso em poucas funções estruturantes. Finalmente o *algoritmo por propagação* processa apenas a fronteira dos objetos da imagem, mas ainda assim computa pixels que não geram resultados.

Forças

- O *algoritmo paralelo* é fácil de implementar, mas não tem eficiência.
- O *algoritmo sequencial* é mais eficiente que o paralelo, mas restringe o seu uso em poucas funções estruturantes.
- A erosão *por propagação* também desperdiça tempo computacional verificando pixels que não vão gerar resultados, além de ter a implementação mais difícil.

Solução

Usar *Propagação Direcional*. Ou seja, consideramos uma direção de propagação na fronteira do *algoritmo por propagação*. Isto permite que processemos somente pixels que gerarão resultados e também apenas as adjacências destes pixels que podem gerar resultados.

A estratégia adotada é considerar a direção em que a erosão ocorre, avaliando esta expressão apenas nesta direção.

Para isto, ∂f_b armazena, além da coordenada x, a direção de propagação da erosão como um inteiro $k \in [1, 8]$, cuja direção correspondente a propagação de x, conforme a Figura 4.

5	6	7
4		8
3	2	1

Figura 4: Direções de propagação

Por exemplo: na Figura 5 o pixel $(2, 3)$ pode ser erodido considerando apenas as direções $3, 4$ e 5 (isto é, $(3, 2), (2, 2)$ e $(1, 2)$).

0	1	2	3	4	5
1	0	0	k	k	k
2	0	0	k	k	k
3	0	0	k	k	k
4	0	0	0	0	0
5	0	0	0	0	0

Figura 5: Exemplo para propagação direcional, onde a primeira linha e a primeira conluna são os índices da imagem

O algoritmo baseado em erosão por propagação usando a informação da direção é dado por (agora ∂f_b contém $[x, d]$, a coordenada x e a direção de propagação d):

Function $[g, \partial g_b] = eroInit(f, b)$
  $\{g$ e ∂g_b são parâmetros de saída$\}$
  $g = f$;
  for all $x \in \mathbf{E}$
    for all $y \in (B + x) \cap \mathbf{E}$
      $\{$ d é a direção de y em $B + x$ $\}$
      if $g(x) > f(y) \dot{-} b(y - x)$
        $g(x) = f(y) \dot{-} b(y - x)$;
        if $x \notin \partial g_b$, $\quad set\_in(\partial g_b, [x, d])$;

Function $[g, \partial g_b] = eroDir(f, b, \partial f_b)$
  $\{g$ e ∂g_b são parâmetros de saída$\}$
  $g = f$;
  for all $[x, k] \in \partial f_b$
    for all $y \in (B'[k{-}{-}, k, k{+}{+}] + x) \cap \mathbf{E}$
      if $g(y) > f(x) \dot{-} b(x - y)$
        $g(y) = f(x) \dot{-} b(x - y)$;
        $set\_in(\partial g_b, [y, d])$;

onde $[k{-}{-}, k, k{+}{+}]$ é o subconjunto das direções: $anterior$, k e $posterior$ a k, considerando a ordem horária. Se $k = 1$, então $anterior$ é 8, e se $k = 8$, $posterior$ é 1. Na última linha do código acima, d é $k{-}{-}$, k ou $k{+}{+}$ dependendo da direção do y usado no teste, conforme Figura 4.

Usos Conhecidos

Uma série de operações morfológicas podem usar *propagação direcional*. Erosão, dilatação, fechamento, abertura e esqueleto são apenas alguns exemplos.

A Transformada de Distância Euclidiana (TDE) é outro exemplo, pois pode ser implementada usando-se erosões morfológicas que por sua vez podem ser implementadas usando *propagação direcional*.

A seguir explicaremos os conceitos de TDE usando erosão morfológica. Maiores informações podem ser encontradas em [Zampirolli, 2003].

Shih e Mitchell foram os primeiros a mostrar que a TDE pode ser obtida de forma exata pela erosão morfológica usando uma função estruturante parabolóide b_E aplicada sobre uma imagem binária f com valores 0 e k [Shih and Mitchell, 1992]:

$$\Psi_{dE}(f) = \varepsilon_{b_E}(f).$$

O valor na origem de b_E é zero e nos outros pontos é dado pelo negativo do quadrado da distância Euclidiana à origem.

Uma propriedade da erosão específica para a transformada de distância é a *idempotência*, i.e., se aplicarmos a erosão por b_E novamente, o resultado não se modifica:

$$\varepsilon_{b_E}(\varepsilon_{b_E}(f)) = \varepsilon_{b_E}(f).$$

Por exemplo, a Figura 6 mostra b_E, onde a origem da função estruturante está marcada em negrito. Esta função estruturante pode ser usada para calcular a transforma de distância em imagens onde o maior valor da distância seja 2. Na Figura 7b é mostrado

		-4		
	-2	-1	-2	
-4	-1	**0**	-1	-4
	-2	-1	-2	
		-4		

Figura 6: função estruturante b_E.

a erosão de f por b_E.

0	0	0	0	0	0	0	0	0
0	0	0	k	k	k	0	0	0
0	0	k	k	k	k	k	0	0
0	k	k	k	k	k	k	k	0
0	k	k	k	0	k	k	k	0
0	k	k	k	k	k	k	k	0
0	0	k	k	k	k	k	0	0
0	0	0	k	k	k	0	0	0
0	0	0	0	0	0	0	0	0

(a)

0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0
0	0	1	2	4	2	1	0	0
0	1	2	2	1	2	2	1	0
0	1	4	1	0	1	4	1	0
0	1	2	2	1	2	2	1	0
0	0	1	2	4	2	1	0	0
0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0

(b)

Figura 7: (a) Imagem de entrada f e (b) $TDE(f)$.

Contexto Resultante

A eficiência do padrão direcional pode ser observado na TDE, pois usa de forma iterativa as erosões direcionais. Veja a seguir os desempenhos das implementações desses três algoritmos comparando com os algoritmos do Eggers [Eggers, 1998] e do Ragnemalm [Ragnemalm, 1992]. As imagens testadas são de tamanho 256×256, onde $img1$ é uma imagem contendo um único pixel do fundo colocado no centro da imagem; $img2$ é uma imagem contendo quadrados de tamanhos variados e $img3$ é uma imagem com círculos de tamanhos variados, ambas possuindo 20% de pixels do objeto. Analisando os desempenhos apresentados acima, vemos que os algoritmos $distPro$ e $distDir$ apresentam eficiências semelhantes ao algoritmo do Eggers. Justificamos o desenpenho de $distDir$ inferior ao $distPro$ em alguns casos, pelo fato do aumento da complexidade das estruturas de dados envolvidas, mesmo fazendo menos cálculos por considerar a direção de propagação.

	Rag	Egg	Par	Pro	Dir
$img1$	3.02	1.03	16.27	1.05	1.14
$img2$	5.98	0.47	4.18	0.33	0.50
$img3$	2.78	0.71	3.52	0.40	0.36

Tabela 1: Tempo em segundos do desempenho de diversos algoritmos:
$Rag-$Ragnelmalm [Rag92];
$Egg-$Eggers [Egg98];
$Par-$TDE por erosão paralela;
$Pro-$TDE usando erosão por propagação;
$Dir-$TDE usando erosão por propagação com informação de direção.

Padrões Relacionados

Paralelo [D'Ornellas, 2003]
Sequêncial [D'Ornellas, 2004] e
Por Propagação [D'Ornellas, 2002].

Referências

Birkhoff, G. (1967). *Lattice Theory*. American Mathematical Society, Providence, Rhode Island.

D'Ornellas, M. (2002). A queue-based algorithmic pattern. In *Proceedings of the Second Latin American Conference on Pattern Languages of Programming - SugarLoafPloP*, pages 279–298, São Paulo, SP. Editora do IME-USP.

D'Ornellas, M. (2003). A parallel algorithmic pattern. In *Proceedings of the Third Latin American Conference on Pattern Languages of Programming - SugarLoafPloP*, Porto de Galinhas, PE.

D'Ornellas, M. (2004). A sequential algorithmic pattern. In *Proceedings of the Fourth Latin American Conference on Pattern Languages of Programming - SugarLoafPloP*, Fortaleza, CE. Editora da UFC/Hillside Group/Instituto Atlântico.

Eggers, H. (1998). Two fast Euclidean distance transformations in z^2 based on sufficient propagation. *Computer Vision, Graphics and Image Processing*, 69(1).

Heijmans, H. (1991). Theoretical aspects of gray-level morphology. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):568–581.

Ragnemalm, I. (1992). Neighborhoods for distance transformations using ordered propagation. *Computer Vision, Graphics and Image Processing: Image Understanding*, 56(3).

Shih, F. and Mitchell, O. (1992). A mathematical morphology approach to Euclidean distance transformation. *IEEE Transactions on Image Processing*, 1:197–204.

Zampirolli, F. (2003). *Transformada de distância por morfologia matemática*. PhD thesis, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, Campinas, SP, Brasil.

The Layered Information System Test Pattern

Roberta Coelho, Uirá Kulesza, Arndt von Staa, Carlos Lucena

Software Engineering Laboratory, Computer Science Department
Pontifical Catholic University of Rio de Janeiro - PUC-Rio
e-mail: [roberta, uira, arndt, lucena]@inf.puc-rio.br

Abstract

The object oriented application layer architecture [3, 12] allows the distribution of classes into well defined layers, according to different purposes (business, communication, data access, etc.). Elements from different layers communicate only through interfaces. While this architecture helps to address requirements of many applications, it also creates many new challenges to software testing [9]. Developers must look around for some techniques that help isolate bugs more quickly in this architecture. Test pattern is a technique that can improve the efficiency of the testing process, since, it provides a means to share test construction experience. While design patterns describe interactions between classes and determine the specification of the classes that participate in the solution of a specific design problem, a test pattern defines a configuration of objects needed to test the interactions between classes. Both are intended to guide the construction of a piece of software. The Layered Information System Test Pattern documents a systematic way of testing a layered information system which is based on exercising only the interface defined by each layer.

Intent

The *Layered Information System Test Pattern* proposes a set of test classes to exercise an information system structured according the Layer Architectural Pattern. Each test class exercises the interface of each layer, focusing on the specific concerns/features implemented by a layer. This pattern also allows the execution the unit test of each layer and the integration tests between layers.

Example

This section presents an illustrative example of an information system that supports the management of bank accounts. Figure 1 presents the object-oriented architecture of this information system following the Layer architectural pattern [3, 12]. According to this pattern, the elements from each layer should communicate only through well defined layers` interfaces. The purpose of a layer interface is to define the set of available operations - from the perspective of interacting client layers - and to coordinate the layer response to each operation.

Several design patterns have been proposed to refine each layer of this architecture. Some of them are: the Service Layer Pattern [13], the Data Access Object Pattern [11] and the Persistent Data Collections (PDC) [1].

The example, presented in the Figure 1, focuses on the Business and Data layers of a Bank Information System. There can be a GUI layer on top of them; however, this pattern will just focus on the layers illustrated in Figure 1.

Business and Data layers are defined according to PDC pattern. Nevertheless, different design patterns [1, 11] could be adopted to refine the information system layers, according to the system requirements and the platform used by the application. PDC design pattern [13] refines each layer by filling them with specific classes and interfaces related to business and data access concerns.

Following the guidelines defined in the PDC pattern, the Business layer should provide a Facade [2] to the system functionality, a unique interface for its services. In this example the Facade role is played by the `Bank` class. The Business layer also specifies a set of business collection classes (`ClientRecord`, `AccountRecord`) which defines business rules related to each entity classes (`Client`, `Account`). The business collection classes are also responsible for accessing the services of the Data layer in order to execute persistence operations, such as, insertions, searches, updates, deletions.

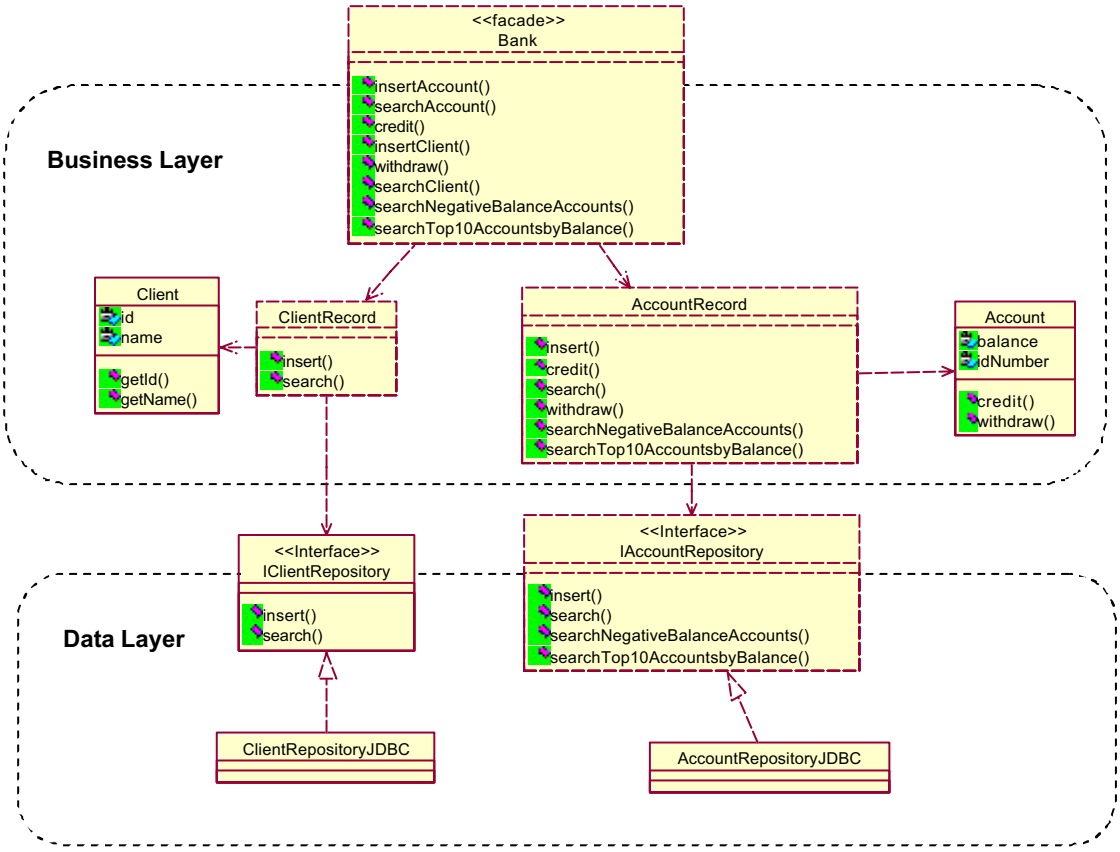


Figure 1. Object-Oriented Design for the Bank information system.

The Data layer interface can be structured in one or more classes. In Figure 1, the Data layer interface is structured in two modules one to each main business entity defined in the business layer (`IClientRepository`, `IAccountRepository`). These interfaces are implemented according to a specific persistence platform, in Figure 1, `ClientRepositoryJDBC` and `AccountRepositoryJDBC` classes implement data access operations related to a specific Entity class using the Java Database Connection (JDBC) API.

Context

Many information systems developed nowadays, define their architecture based on the Layer architectural pattern [3, 12]. This architectural pattern allows the distribution of classes into well defined layers according to different concerns, such as user interface, communication, business and data access. Also, several design patterns [1, 11, 13] have been proposed to refine each layer of this architecture.

Despite those patterns have been widely used, the model for testing systems structured according to this architectural pattern has received few attention and has been few explored. Most tests are limited to test suites and test cases using simple strategies [7,10]. Although those tests are useful, they fall short in the role of a general organizational for automated testing. What is required is a higher level of abstraction, a test pattern that can be reused wherever a layered architecture is adopted.

Problem

Due to the lack of well defined test patterns, developers and test engineers have applied adhoc or not well defined strategies during system testing. Some examples of common test strategies which have been adopted during system testing are the following:

- execution of adhoc manual tests in the user interface layer;
- specification of unit tests to some classes of the system which are chosen using no systematic strategy;
- implementation of one test class to every class of the system (Test Driven Development – Extreme Programming practice [7]).

Although, these test strategies can eventually help system debugging, there are many disadvantages associated to such strategies, such as:

- the difficulty of finding the exact faulty code that causes a system failure. Sometimes, during manual tests complex sequence of actions are performed, which can not be repeated.
- high cost and effort necessary to reexecute manual tests;
- a great amount of resources and effort can be wasted due to the codification of many unit tests that will not be effective during system testing;
- since the classes to be tested are chosen without good selection criteria, important system functionalities may be forgotten during testing.

The development of an information system typically addresses different concerns, such as, user interface, distribution, business and data access. The lack of well defined strategies to test an information system can bring several problems to system quality and additional costs to

the software development. A recurring problem in the context of layered information systems is how to systematically define automatic tests to verify the functionality of each layer in isolation and in collaboration with other layers.

Forces

The following forces influence solutions to this problem:

- *Importance of Tests*: Software testing adds value to a system by revealing its faults. It produces evidence that a pre-release reliability goal has been met.
- *Resource Limitation*: Testing is an expensive process. Test process should continue until a reliability goal is attained, but mot of the time it continues until available test resources have been expended.
- *Minimum set of Test-Cases*: We would like to reduce the cost of testing process without decreasing test quality. We would like to define a minimum set of test cases that would exercise system main components and functionalities.
- *Separation of Concerns*: Developers should focus on each specific layer when testing the system. Besides, they should be able to test each layer independent from the others.
- *Test Class Modularity*: Each test class should verify a well defined set of functionalities provided by one specific layer.
- *Test Robustness*: The test classes should be resilient to internal changes in the implementation of the layer classes.
- *Proximity between Fail and Fault*: Automatic tests should make it easier to come across system failures as well as to localize the faults that had caused them.

Solution

Create unit tests to exercise only the interface defined by each layer. Each test class focuses on the test of specific concerns/features implemented by a layer. Furthermore, the test code responsible for verifying all the services provided by a layer can be modularized in one or more test classes.

To allow the test of one layer at a time, this pattern adopts auxiliary classes, called mock objects [8]. A Mock Object is used by a test to replace a real component (or a set of components) on which the system under test depends. Typically, mock objects fakes the implementation either by returning hard coded results or results that were pre-loaded by the test [8].

Since the tests defined by the Layered Information System Test Pattern exercises only the interface of each layer, and there is not a one-to-one relationship between the classes that comprises the interface and the test classes, this pattern can be used to test any layered information system no matter the design pattern or design strategy used to refine the layers.

Structure

Figure 2 illustrates the structure of the Layered Information System Test Pattern. It has three participants:

- **BusinessTest**: this class contains all methods that test a set of functionalities provided by the Business Layer Interface and are related according to one specific criterion. This criterion can be a set of operations related to a business entity or to a business service.
- **BusinessRepositoryTest**: implements test methods to all methods provided by a Repository interface. The implementation of these test classes focus on the testing of specific data repository functionality related to insertion, searching, update and database operations. Each test method implements a test case which verifies a successful or an error condition from a specific repository method.
- **MockRepository**: this class fakes the implementation of a specific BusinessRepository. Thus, this auxiliary class enables the unit test of the business layer.

All Business Layer`s operations can be structured in one single interface or a set of interfaces [2]. The purpose of the `BusinessTest` classes is to modularize the Business Layer tests according to each business entity manipulated by its operations or according to each business services implemented by such operations. For example, there can be one `BusinessTest` class to exercise the set of operations related to a business entity or a business service.

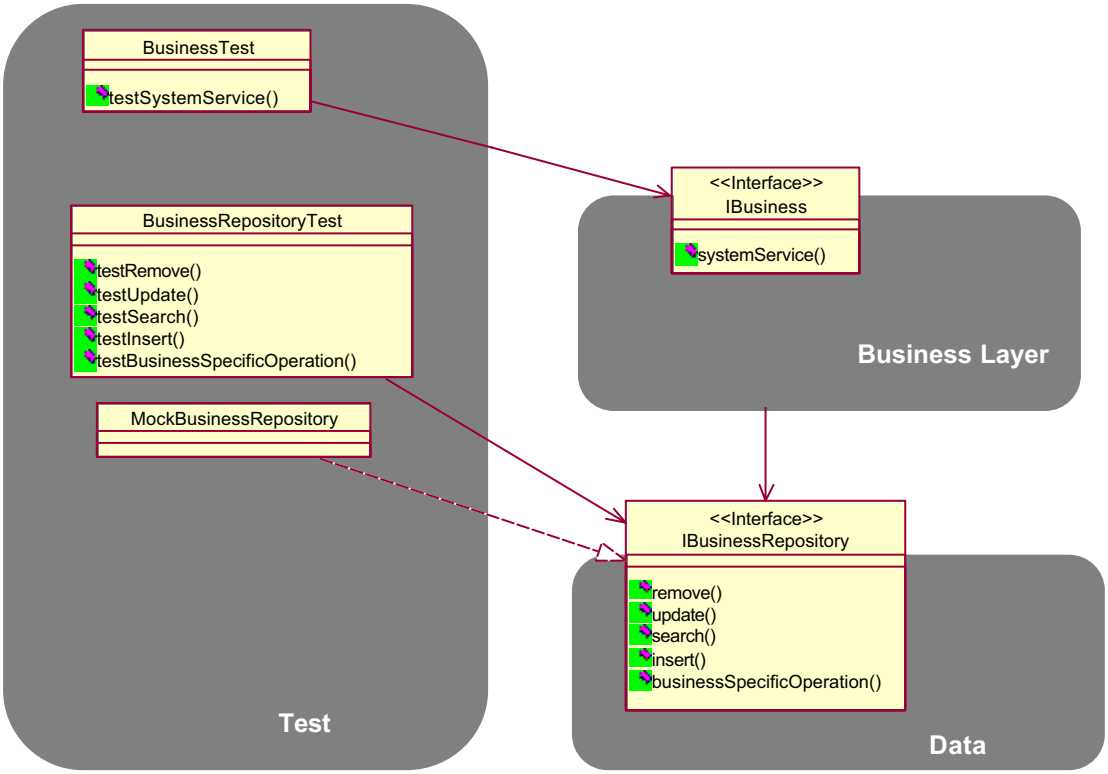


Figure 2. The Static View of the Layered Information System Test Pattern.

The `BusinessTest` classes contain a test method to each successful and error condition of each method from the Business Layer. Most of the time developers focus on testing successful conditions and forget the error ones, which are as important as the former. If we define only one class to test all successful and error conditions of Business Layer methods, the resulting test class will probably contain too many lines of code which can impact on test maintainability.

The Data Layer will also be tested through a set of classes which exercises its interface. Each Data Layer test class concerns with one specific business repository accessible through the Data Layer`s interface. The `BusinessRepositoryTest` classes, illustrated in Figure 2, are the ones responsible for testing the each business repository.

Since each layer delegates services to the lower layer the only way to test Business Layer without the passing through Data Layer is to delegate data services to the `MockBusinessRepository` class, which fakes the implementation of a real `BusinessRepository` class either by returning hard coded results or results that were pre-loaded by the test.

The `MockBusinessRepository` classes allow the `BusinessTest` classes to concentrate on testing Business Layer own code. Therefore, the integration test of those two layers is performed when Business Layer delegates services to the real repositories instead to the mock classes.

Dynamics

This pattern allows the execution of three types of tests: Business Layer unit test, Data Layer unit test, and integration test of Data and Business Layers.

Figure 3 illustrates the sequence of method calls performed during Data Layer unit test. Firstly, an instance of `BusinessRepository` class is created during the initialization of `BusinessRepositoryTest` class (steps 1 and 2), Secondly, a test method is called, for example, `testInsert()` (step 3), then, `setup()` method is called – a private method responsible for any configuration and initialization common to all test methods (step 4). Finally, `BusinessRepository` methods are called (steps 5, 8 and 9) and assert operations are executed to compare expected results with returned results (steps 7 and 10).

Figure 4 represents an integration test comprising the Business Layer and the Data Layer. It illustrates the sequence of method calls performed when the Business Layer is tested in collaboration with the Data Layer. Firstly, the `BusinessTest` class creates the classes that implement the Business and Data layers. In the Figure 4, this is illustrated through the instantiation of classes that implement the `IBusiness` and `IBusinessRepository` interfaces (steps 2 and 3). After that, different test methods can be executed in order to exercise the functionalities implemented by the Business Layer. Figure 4 illustrates the execution of the `testSystemService()` method, which calls a business method (step 6) and uses an `assert()` method (step 7) to to compare returned results with expected results.

Figure 5 illustrates the sequence of method calls performed when testing the piece of functionality embedded in the Business Layer. This type of test, as distinct to the integration test described previously, exercises a single layer. Since Business layer depends on the services provided by Data Layer, those services should be emulated by a fake implementation of such layer, a mock object. In the Figure 5, the instantiation of Business Layer is represented

by the creation of the `IBusiness` object (step 3). As we can see, this `IBusiness` object receives an instance of `MockBusinessRepository` (step 2), which will simulate the repository implementation. Since the mock object implements the same interface of a real repository and the `IBusiness` object does not know that it is dialing with a "fake" implementation of a repository. Finally, the test methods are executed the same way as described in Figure 5.

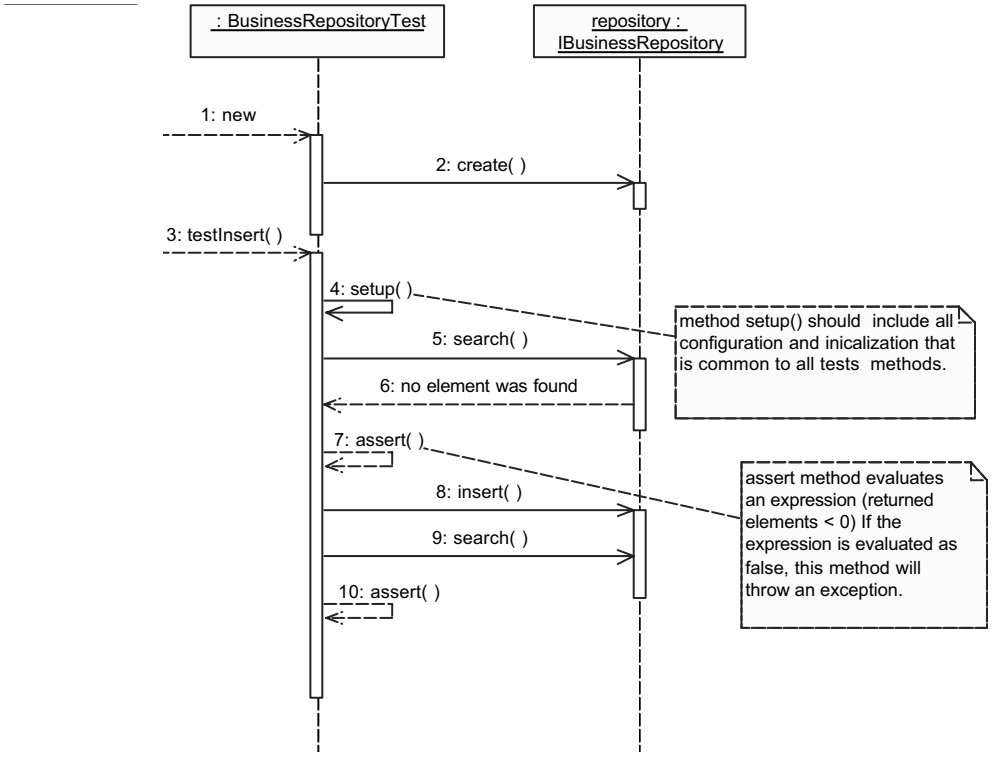


Figure 3. Dynamic View of the Data Layer unit test.

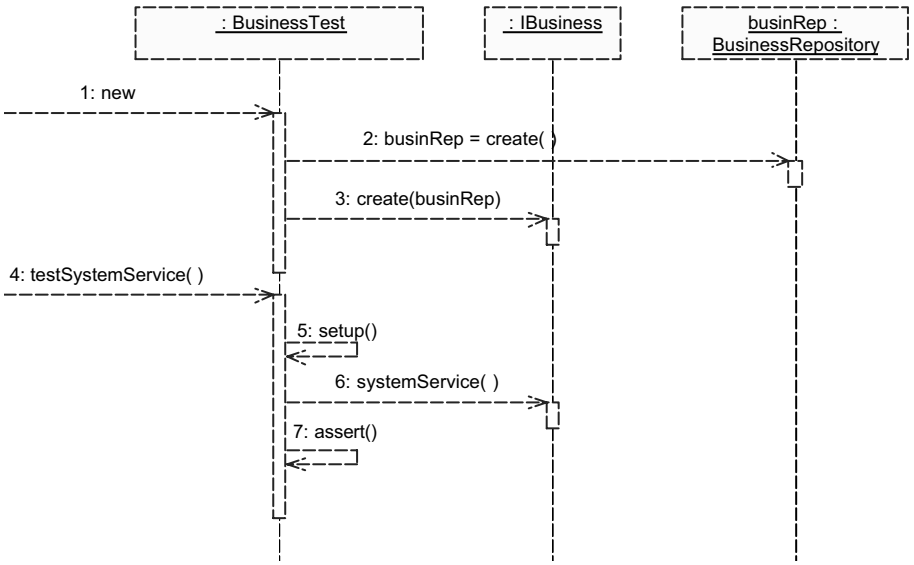


Figure 4. Dynamic View of the integration test of the Data Layer and the Business Layer.

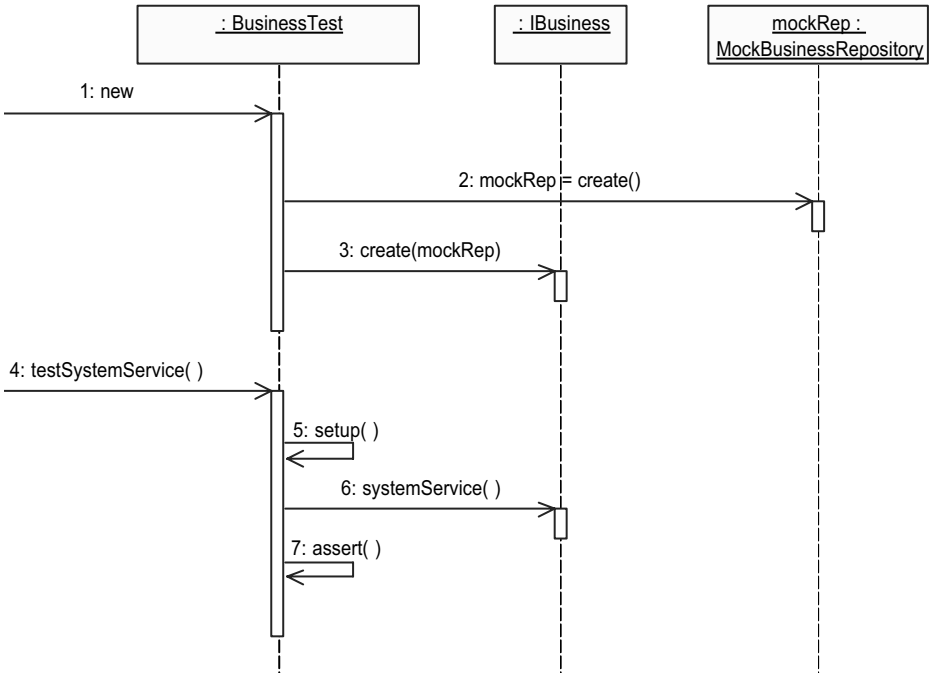


Figure 5. Dynamic View of Business Layer unit test.

Example Resolved

Figure 6 presents the use of the Layered Information System Test Pattern for the bank information system illustrated previously. Two classes, `AccountRepositoryTest` and `ClientRepositoryTest`, are specified to enable the testing of the data access classes. These classes are implemented based on the method signatures defined in the `IAccountRepository` and `IClienteRepository`, respectively. This allows to reuse them in case the system developers need to provide new data access classes to a different persistency platform.

The test of the Business Layer for the example of the bank information system is supported by the `AccountOperationsTest` and `ClientOperationsTest` classes. Each of these classes implements a set of test methods related to a specific entity class. Also, as we can see in the Figure 6, these classes are codified based only on the business methods provided by the Bank facade class. Thus, internal changes in the implementation of these services do not affect the test classes.

Finally, two mock auxiliary classes, `MockAccountRepository` and `ClientAccountRepository`, are presented in the Figure 6. They represent alternative implementations of the data access classes. They are used when it is required to test the Business layer functionality individually.

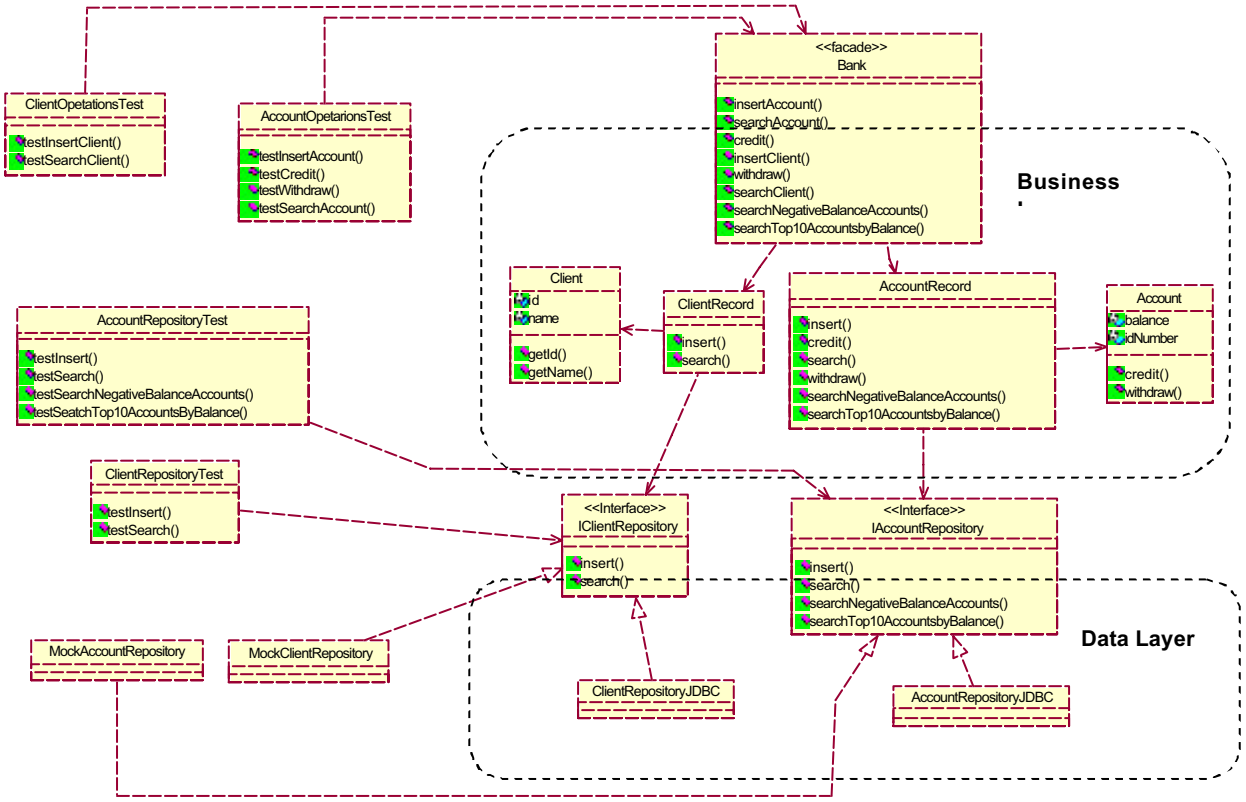


Figure 6. An information system and its corresponding test classes

Consequences

The Layered Information System (LIS) Test Pattern maintains the following consequences:

- *Separation of Concerns*. The pattern defines an individual test to each layer of an information system. LIS test pattern focus on the testing of individual services.
- *Test Class Modularity*. The testing code is modularized using different test classes. Each test class focus on the verification of a well defined and limited set of functionalities provided by a specific layer. It improves the readability and maintainability of the test classes.
- *Test Robustness*. Since test classes depend only on the layer interface, they are no effected due to implementation changes inside a layer.
- *Increase in Cost of System Development*. Although there is a cost associated to the implementation of the layered information system test pattern, the systematization of the test activity can reduce its cost if compared with other approaches, such as adhoc tests and unit test of every class. Code generation tools can even reduce test costs since they can generate the overall structure of many test classes. Moreover, if a developer decides to skip test activities, afterwards, the system will be buggy and will consequently cost more time and money to be fixed.

- *Proximity between Failure and Fault*. LIS test pattern defines individual tests to each layer. When a fault is detected by a test case, such minimum set of tests cases allows the developer to identify in which layer is the fault, but cannot diagnosis which specific class causes the failure.
- *Increase in the number of classes*: a negative consequence of this testing solution is the increase in the number of classes to be maintained. However, this Test Pattern allows the execution of automated tests along the iterations which would require high cost and effort to be reexecuted manually. Although this pattern suggests fewer test classes than Test Driven Development (TDD) agile practice (one unit test per class) it is as effective as TDD. Since the classes to be tested are chosen according to a specific criterion, important system functionalities is not forgotten during testing.

Known Uses

The Layered Information System Test Pattern has been used during the development of two Java information systems in Recife, Brazil. A general description of these systems is given below.

- A system for managing real estate. This system allows the register of real estate and the management of tax charging related to them. It was implemented in the J2EE platform.
- A system that supports the management of market activities. The system allows the register of market activities and the management of tax charging related. It was also implemented in the J2EE platform, including the use of the Enterprise Java Bean technology.

See Also

A few test patterns have already been proposed. Gerard Meszaros [4, 5] has proposed two Test Pattern languages, one for setting up XUnit test features - which describes key techniques for addressing the issues around test fixture management, and the other for automating testing of indirect inputs and outputs using XUnit.

Some design patterns for using Mock Objects have been proposed as well, some of them are the following:

- *Mock Object*: a basic mock pattern that allows for testing a unit in isolation by "faking" the communication between collaborating objects.
- *Mock Object Factory*: a way of creating mock objects using existing factory methods.
- *Mock Object via Delegator*: a pattern that creates a mock implementation of a collaborating interface in the test class or mock object.

Implementation

We describe below some guidelines for implementing the Layered Information System (LIS) Test Pattern. The following code examples are related to an information system for managing bank accounts presented in previous sections. They are written using the Java programming language and the JUnit test framework [6]. However, the LIS Test Pattern can be implemented in other platforms, by following the guidelines we present below.

Step 1: Prepare the Entity classes to help the codification of test classes.

Every test method needs to evaluate the data sent or received from the methods being tested. In the context of information systems, the information manipulated are, typically, the content embedded in entity classes. Thus, before starting the implementation of test classes, it is important to define a way to compare two instances of the same Entity class. A well known way to compare two instances of a class is through the a method `equals()` that receives an instance of the same class and returns true if the argument contains the same attributes values as the class being called or false otherwise.

In the information system for the management of bank accounts, for example, the `Account` class must define its `equals()` method in order to compare its attributes `idNumber` and `balance` with the same attributes of other instance.

```
public class Account {
      private long idNumber;
      private double balance;

      public Account(long idNumber, double balance){
      this.idNumber = idNumber;
      this.balance = balance;
      }

      ...

      public boolean equals(Object anotherInstance){
         Account anotherAccount = (Account) anotherInstance;

         if ( this.idNumber == anotherAccount.idNumber &&
              this.balance == anotherAccount.balance){
           return true;
         }else {
           return false;
         }
      }
}
```

Step 2: Define a BusinessRepositoryTest class.

A BusinessRepositoryTest class must define test methods to verify the functionality provided by a data access class (or data repository class) which are specified in the business repositories interfaces.

As mentioned in the Structure Section, a BusinessRepositoryTest class has many responsibilities, such as: (i) to create an instance of a data access class to be tested; (ii) to define a method that performs every configuration and initialization necessary to run the test; and (iii) to specify different test methods to each method provided by the data access class to be tested.

Each BusinessRepositoryTest class must define different test methods to each existent method of the data access classes. These test methods must verify the successful and error conditions, using different argument types and values and handling different types of exceptions.

In order to minimize effort, the search methods - of the data access classes - can be used to support the test of the other methods. For example, the test method of insert operations can, previously, search the object be inserted to verify if it does not already exist in the repository. Also, the test methods of delete and update operations should use the search method whenever they need.

Below, we present the partial code of a BusinessRepositoryTest class in the context of the banking system, responsible to test the functionality of an `IAccountRepository` instance.

```
public class AccountRepositoryTest extends TestCase {
      private IAccountRepository accountRepository;

      public AccountRepositoryTest(String name){
         this.accountRepository = new AccountRepositoryJDBC();

         // Additional common configurations before to execute
         // all the test methods
         ...

      }

      // JUnit standard method to be executed before every test method
      protected void setUp() {
         ...
      }

      public void testInsertAccount() {
            try {
              Account account = new Account(123, 500);
              accountRepository.inserir(account);

              Account accountSearched = accountRepository.search(123);
              assertEquals(account, accountSearched);

            } catch (Exception e) {
               fail("Exception not expected:" + e);
            }
       }
       public void testInsertAlreadyExistentAccount() {
            try {
               Account account = new Account(123, 500);
               accountRepository.inserir(account);
               fail("System did not throw exception!!!");
```

```
            Account accountSearched = accountRepository.search(123);
            assertEquals(account, accountSearched);

          } catch (AlreadyExistsObjectException e) {
             System.out.println("OK: Exception expected!!!");

          } catch (Exception e) {
             fail("Exception not expected:" + e);
           }
       }
       ...
}
```

Step 3: Define a MockBusinessRepository class.

The MockBusinessRepository classes simulate the behavior of BusinessRepository classes in order to allow the unit test of the Business Layer.

In order to fake the behavior of a real repository the MockBusinessRepository classes can use an internal data structure (like a hash table or a vector) that is able to store the business objects. The Mock classes must implement the data access interfaces. Each method described in these interfaces uses the internal data structure.

A partial code of the `MockAccountRepository` class is presented below. It uses a hash table to store the business objects manipulated by the mock.

```
public class MockAccountRepository implements IAccountRepository {
    private Map accounts;
    public MockAccountRepository(){
      this.accounts = new Hashtable();
    }

    public void insert(Account account)
                throws AlreadyExistentObjectException, ... {
      if (this.accounts.containsKey(new Long(account.getIdNumber()))){
          throw new AlreadyExistsObjectException
                                ("Object already exists");
      }else {
         this.accounts.put(new Long(account.getIdNumber()), account);
      }
    }
    public Account search(long idNumber)
                    throws InexistentObjectException {
      Account account = null;

      if (this.accounts.containsKey(new Long(idNumber))){
        account = (Account) this.accounts.get(new Long(idNumber));
      }else {
        throw new InexistentObjectException "Object does not exist");
      }
      return account;

     }
     ...
}
```

Step 4: Define a BusinessTest class.

A BusinessTest class verifies the functionality provided by Business Layer. Different BusinessTest classes should be defined for each system.

This class contains all methods related to a business entity or to a business service. In this example each test class must focus on the testing of all Facade operations related to a business entity. Moreover, each test method defined must verify different execution conditions of the method under test, such as: (i) the correct execution of business rules; and (ii) the incorrect execution which throws business exceptions.

In the example presented in *Solved Example* Section, two different BusinessTest classes were be specified: one responsible for testing the functionalities related to the `Account` class and the other responsible for testing the functionalities related to `Client` class. Below we present the `AccountOperationsTest` class, responsible for testing the methods in the `Bank` facade class related to the Account business class. We can also observe that the `AccountOperationTest` class constructor allows two different configurations depending on the kind of test that will be executed: (i) in case we want to perform integration tests, the Data layer will use the system data access classes; and (ii) in case we want to perform unit tests in the Business Layer, the Data layer should be replaced by a mock object in the test method. In a more realistic implementation of BusinessRepositoryTest classes, the parameter `integrationTest` should be loaded from a configuration file.

```
import junit.framework.TestCase;

public class AccountOperationsTest extends TestCase {
   private Bank bank;
   private boolean integrationTest = true;

   public AccountOperationTest(String name){
       this.bank = Bank.getInstance();

       AccountRecord accountRecord = null;
       ClientRecord = clientRecord = null;
       if (integrationTest){
          accountRecord = new AccountRecord(new AccountRepositoryJDBC());
          ...
       } else {
          accountRecord = new AccountRecord(new MockAccountRepository());
          ...
       }
       this.bank.setAcccountRecord(accountRecord);
       ...
    }

    // JUnit standard method to be executed before every test method
    protected void setUp() {
       ...
    }

    public void testCreditAccount() {
      try {
        Account account = new Account(123, 500);
```

```
        bank.insertAccount(account);

        bank.credit(123, 200);

        Account accountSearched = bank.searchAccount(123);
        assertEquals(new Account(123, 700), accountSearched);

      } catch (Exception e) {
        fail("Exception not expected:" + e);
      }
    }

    public void testWithdrawAccount() {
       try {
         Account account = new Account(456, 500);
         bank. insertAccount(account);
         bank.withdraw(456, 200);

         Account accountSearched = bank.searchAccount(456);
         assertEquals(new Account(456, 300), accountSearched);

      } catch (Exception e) {
         fail("Exception not expected:" + e);
      }
   }
   ...
}
```

Acknowledgments. We would like to give special thanks to Carlo Giovano, our shepherd, for his important comments, helping us to improve our pattern. This work has been partially supported by CNPq under grant No. 150678/2004-7 for Roberta de Souza Coelho and grant No. 140252/2003-7 for Uirá Kulesza, and by FAPERJ under grant No. E-26/151.493/2005 for Uirá. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0 and by the art. 1st of Decree number 3.800, of 04.20.2001.

References

1. D. Alur, D. Malks, J. Crupi. Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, 2nd edition, 2003.
2. E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995..
3. F. Buschmann et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley Sons, 1996.
4. G. Meszaros. A Pattern Language for Automated Testing of Indirect Inputs and Outputs using XUnit, Proc. of the 11th Conference on Pattern Languages of Programs (PLoP2004), September 2004, Monticello, USA.
5. G. Meszaros. A Pattern Language for Setting up XUnit Test Fixtures. Proc. of the 11th Conference on Pattern Languages of Programs (PLoP2004), September 2004, Monticello, USA.
6. JUnit Framework, http://www.junit.org.

7. K. Beck, Extreme Programming Explained, Addison-Wesley, 2000
8. M. Brown and E. Tapolcsanyi, Mock Object Patterns, Proceeding of the PLOP 2003, September 2003, Monticello, USA.
9. M. Donat, Debugging in an Asynchronous World, ACM Queue 1(6), 2003, pp. 23-30.
10. M. Fowler, A UML Testing Framework. Software Development Magazine. April, 1999
11. M. Fowler, et al. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.
12. S. Ambler. Building Object Applications that Work. Cambridge University Press and Sigs Books, 1998.
13. T. Massoni, Vander Alves, Sergio Soares, and Paulo Borba. PDC: Persistent Data Collections pattern. In First Latin American Conference on Pattern Languages Programming SugarLoafPLoP, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

Secrecy with Session Key: Um padrão de criptografia para evitar ataques de criptoanálise por textos cifrados conhecidos

Windson Viana[1], José Bringel Filho[2], Rossana Andrade[1,2]

[1]Mestrado em Ciência da Computação – Universidade Federal do Ceará (UFC)

[2]Centro Nacional de Processamento de Alto Desempenho no Nordeste

{windson, bringel, rossana}@lia.ufc.br

Abstract. *This paper presents an extension of the Tropyc pattern language, describing a solution for the confidentiality problem applying ciphertext-only cryptoanalysis of techniques. This problem is aggravated when messages contain known parts by attackers, which facilitates cipher discovery. This ciphered messages with known message parts allows then break the cipher and, as a consequence, the confidentiality of the sent messages.*

Resumo. *Este artigo apresenta um padrão que pode ser utilizado por desenvolvedores de sistemas de criptografia orientados a objetos. O padrão complementa a linguagem de padrões Tropyc, definindo soluções para o problema de quebra de confidencialidade através de técnicas de criptoanálise de ciphertext-only. Este problema se agrava quando as mensagens possuem partes conhecidas pelo atacante, o que facilita a descoberta da cifra. A catalogação, em conjunto com as partes conhecidas das mensagens, permite a quebra da cifra e, em conseqüência, da confidencialidade das mensagens enviadas.*

1. Introdução

A linguagem de padrões Tropyc [Braga 1999] apresenta padrões de soluções criptográficas identificadas a partir de sistemas computacionais que apresentam requisitos de segurança. O primeiro padrão apresentado da linguagem Tropyc é o GOOCA - *Generic Object-Oriented Cryptographic Architecture*, a qual consiste em uma arquitetura genérica para o desenvolvimento de sistemas criptográficos flexíveis e reutilizáveis, utilizando o paradigma da orientação a objeto.

A partir de GOOCA são apresentados padrões de segurança **i**dentificados de acordo com os objetivos primários da criptografia (i.e., confidencialidade, integridade, autenticidade e não-repúdio [Stallings 1999]), bem como a combinações entre eles, descrevendo uma linguagem de padrões fechada ao domínio de segurança.

Para fornecer o serviço de confidencialidade aos sistemas computacionais, Tropyc define o padrão *Information Secrecy*. Este padrão utiliza algoritmos de criptografia simétrica ou assimétrica para prover a confidencialidade das mensagens trocadas entre dois pontos comunicantes. Para cifrar ou decifrar as mensagens, é necessário uma chave secreta ou chaves públicas compartilhadas entre as partes comunicantes. O padrão *Information Secrecy* é ainda combinado com os padrões Tropyc *Secrecy with Sender Authentication, Secrecy with Signature, Secrecy with*

Integrity e *Secrecy with Signature with Appendix*, para a formação de novos padrões com propósitos de segurança.

Entretanto, o padrão *Information Secrecy* não considera o problema originado através de ataques de *ciphertext-only* [Biryukov and Kushilevitz], que permite a quebra da confidencialidade. Nesse tipo de ataque, o invasor somente tem acesso ao texto cifrado, porém ele deduz o texto original ou a chave através de técnicas de criptoanálise. Um exemplo deste ataque é apresentado em [Fluhrer, 2001], no qual a quebra do algoritmo RC4 é realizada.

Este artigo descreve então um padrão, chamado *Secrecy with Session Key*, que apresenta uma solução para proteger a comunicação contra ataques desse tipo.

Na descrição do padrão foram utilizados termos comumente conhecidos na literatura relacionada à criptografia. Por exemplo, Alice e Bob identificam as partes comunicantes, por sua vez, Eve corresponde ao atacante ou criptoanalista que deseja recuperar as informações trocadas entre Alice e Bob, conforme ilustrado na Figura 1.

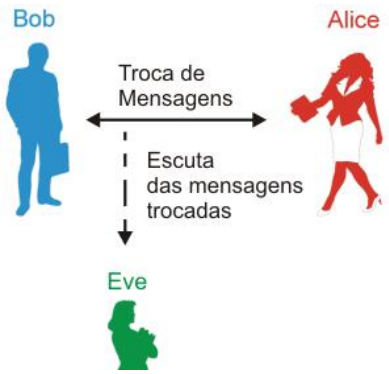


Figura 1. Comunicação entre duas partes comunicantes com a presença de um atacante (adaptado de [Stallings 1999]).

2. Contexto

Alice deseja enviar a Bob mensagens de forma segura. Eles utilizam chaves e algoritmos criptográficos previamente combinados para cifrar e decifrar as mensagens. Entretanto, Eve pode obter acesso às mensagens cifradas e, através de técnicas de criptoanálise de *ciphertext-only*, recuperar as informações transmitidas entre eles.

3. Problema

Como Alice e Bob podem reduzir a possibilidade de Eve recuperar as informações transmitidas através de técnicas de criptoanálise de *ciphertext-only*?

4. Forças

– A recuperação por Eve das informações é facilitada quando a comunicação é intensa (i.e., muitas mensagens trocadas com a mesma chave), quando as mensagens trocadas são curtas (i.e., possuem poucos caracteres) e/ou possuem um formato específico conhecido (e.g., documentos XML, arquivos de imagens e de processadores de texto).

– Será dispendioso Eve, mesmo recuperando todas as mensagens transmitidas no canal, descobrir a(s) chave(s) utilizada(s) para cifrar e decifrar, assim como as informações encriptadas;

– O mecanismo de geração e estabelecimento de *session keys* deve seguro de maneira a evitar que Eve, mesmo capturando as mensagens trocadas durante o processo, não consiga recuperar a chave gerada.

– O custo (e.g., computacional, financeiro) para Eve recuperar as chaves ou quebrar a cifra é maior do que o custo da informação transmitida.

5. Solução

Alice e Bob não devem utilizar sempre a mesma chave ou pares de chaves, no caso da criptografia assimétrica, para cifrar e decifrar as mensagens. Sendo assim, deve ser estabelecida uma chave válida por um determinado período, denominada de chave de sessão (*session key*), além de ser necessário a combinação prévia entre Alice e Bob dos mecanismos para a geração e estabelecimento destas chaves, como por exemplo o algoritmo criptográfico Diffie-Helman [Diffie and Hellman, 1976].

Ao iniciar o processo de comunicação entre Alice e Bob, deve ser verificada a validade da chave de sessão utilizada. Sendo assim, caso o período de validade desta chave esteja expirado, Alice deve executar novamente os mecanismos de geração e estabelecimento de chaves de sessão. A nova chave de sessão deve ser utilizada para cifrar e decifrar as mensagens enviadas e recebidas de Bob, respectivamente. Por fim, para cifrar e decifrar as mensagens, Alice e Bob devem utilizar o padrão *Information Secrecy* da linguagem Tropyc com a chave de sessão estabelecida.

A Figura 2 apresenta o padrão *Secrecy with Session Key* documentado neste artigo, que complementa a linguagem de padrões Tropyc, bem como os novos relacionamentos e padrões que surgiram a partir dele. As arestas contínuas representam as dependências originais entre os padrões Tropyc, enquanto que as tracejadas descrevem as novas dependências entre os padrões Tropyc e o *Secrecy with Session Key*.

Os padrões representados por retângulos cinza com a borda tracejada, representam os padrões Tropyc que, através do relacionamento destes com o padrão *Secrecy with Session Key*, podem dar origem a novos padrões (e.g., *Secrecy with Session Key with Sender Authentication*).

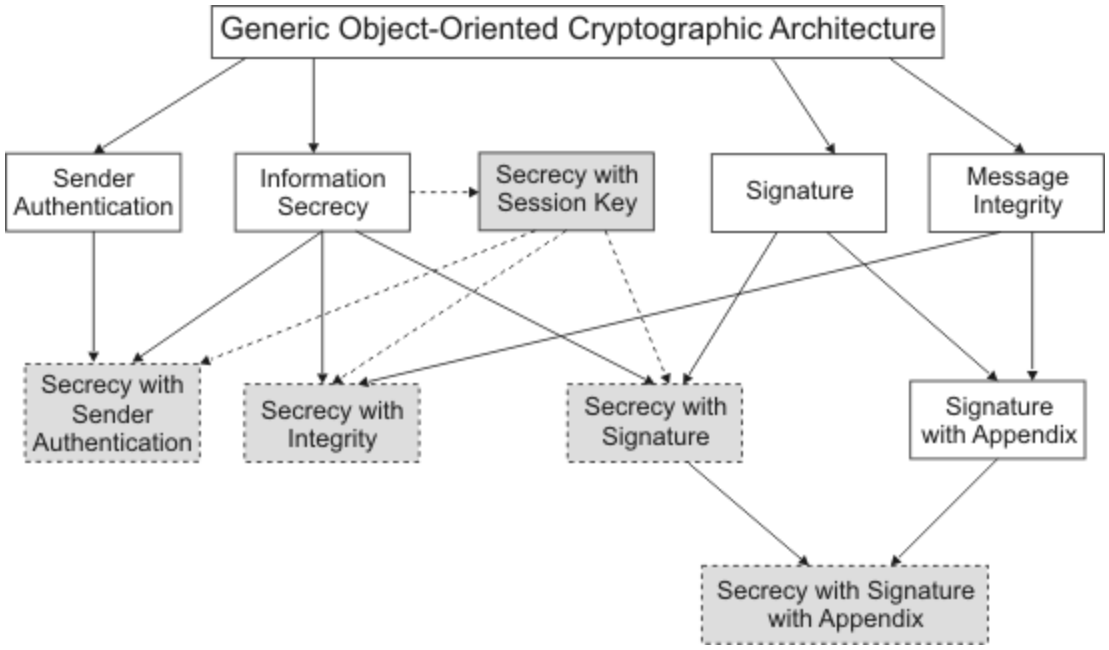


Figura 2. Padrões do Tropyc e o padrão *Secrecy with Session Key* (adaptado de [Braga, 1999]).

6. Conseqüências

- A utilização de mecanismos para a geração e o estabelecimento das chaves de sessão aumenta o tempo de processamento e o número de mensagens trocadas entre Alice e Bob;
- Períodos curtos de validade das *session keys* adicionam *overhead* de processamento e de troca de mensagens;
- Períodos longos de validade das *session keys* facilitam a ocorrência de ataque de *ciphertext-only*.

8. Dinâmica

A Figura 3 ilustra o processo de comunicação entre Alice e Bob, utilizando um diagrama de seqüência, onde é necessário que as mensagens (parâmetro msg) sejam transmitidas de forma segura. Neste diagrama, as classes envolvidas (e.g., codificador, Alice) são as mesmas utilizadas na ilustração do padrão GOOCA da linguagem Tropyc [Braga 1999] e os seguintes passos são executados:

- 1º Passo: antes de dar início à transmissão, Alice verifica a validade da chave de sessão;
- 2º Passo: caso seja constatado que a chave está expirada, Alice, através dos mecanismos de geração e estabelecimento de chave de sessão (SK), irá gerar uma nova SK;
- 3º Passo: caso o 2º Passo tenha sido executado, a chave gerada SK será estabelecida entre Alice e Bob;

– 4º e 5º Passos: a partir desse momento, Alice e Bob utilizam as classes Codificador e Decodificador (que implementam o padrão *Information Secrecy*), além da SK, para cifrar e decifrar as mensagens a serem enviadas.

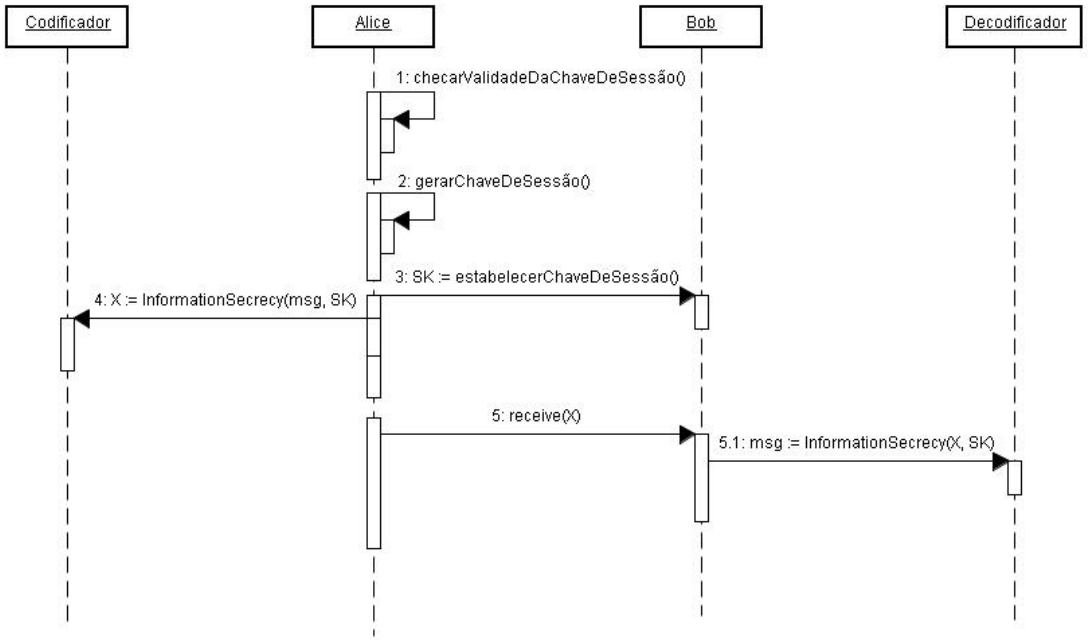


Figura 3. Dinâmica do Padrão *Secrecy With Session Key*.

9. Implementação

Os mecanismos de geração e estabelecimento de chaves de sessão podem ser implementados de duas maneiras distintas: com segredo inicial compartilhado ou sem segredo inicial compartilhado.

No caso da geração e estabelecimento de chaves de sessão com segredo inicial compartilhado, Alice e Bob possuem uma informação inicial previamente distribuída sem o conhecimento de Eve. Essa informação inicial é utilizada nos mecanismos para gerar e estabelecer a nova SK, por exemplo, podem ser utilizados mecanismos de desafio-resposta e de geração de números pseudo-aleatórios [Menezes 1996].

Já no estabelecimento de chave de sessão sem segredo inicial, Alice e Bob utilizam mecanismos de criptografia assimétrica. Neste caso, os padrões *Information Secrecy* e *Secrecy with Signature with Appendix* podem ser utilizados para o estabelecimento da chave de sessão. Além disso, é possível utilizar protocolos de distribuição de chaves, tais como o Diffie-Helman [Diffie and Helman 1976].

10. Padrões Relacionados

– O padrão *Strategy* [Wolfgang 1995] pode ser utilizado para implementar o mecanismo de escolha dos algoritmos de geração e estabelecimento de chaves a serem utilizados pelas partes comunicantes;

– O padrão *Secrecy with Session Key* pode ser combinado a outros padrões da linguagem Tropyc (*Signature, Message Integrity, Sender Authentication, Signature with Apendix*) visando criar padrões que tratam este problema somado ao escopo

original do padrão, como aconteceu com o padrão *Information Secrecy* ilustrado na Figura 2;

- A linguagem de padrões para Gerenciamento de Chaves Criptográficas [Lehtonen and Pärssinen 2002] pode ser utilizada para realizar o gerenciamento de chaves comuns (padrão *Common Key Management*), a geração de chaves criptográficas (padrão *Cryptographic Key Generation*) e a troca de chave de sessão utilizando chaves públicas (padrão *Session Key Exchange With Public Keys*).
- A linguagem apresentada em [deSouza and Matwin 2001] trata de problemas relacionados à comunicação segura na arquitetura cliente-servidor. Os padrões *Public/Private Key Generation*, *Session Key Generation*, *Session Key Exchange* e *Data Encryption/Decryption* podem ser usados em conjunto ao padrão apresentado neste artigo.

11. Usos Conhecidos

A utilização de *session keys* é muito comum em sistemas baseados em criptografia de chave pública e que utilizam a criptografia simétrica para cifrar e decifrar mensagens, por exemplo, o protocolo SSL (*Security Socket Layer*) [Stallings 1999] e aplicações que utilizam PGP (*Pretty Good Privacy*) [RFC 2440 1998]. Por sua vez, o protocolo SSL é utilizado em sites comerciais de venda ou oferta de serviços com a finalidade de realizar a troca segura de informações entre o cliente (i.e., browser web) e o servidor, o qual faz uso de chaves de sessão.

Além disso, este padrão também é encontrado no processo de autenticação de estações móveis nos sistemas de comunicação móvel GSM (*Global System for Mobile Communications*) e GPRS (*General Packet Radio Service*) [Watkins 2000]. Nesses sistemas, é estabelecida uma chave de sessão entre o núcleo da rede e a estação móvel (i.e., aparelho celular) que será utilizada para cifrar a voz e os dados que trafegam na rede.

Referências

Braga, A. M.; Rubina C. M. F.; Dahab, R. Tropyc: A Pattern Language for Cryptographic Software. p. 1-27, Jan. 1999.

Diffie, W.; Hellman, M.E. New directions in cryptography. IEEE Trans. Inform. Theory, 1976. Disponível em http://citeseer.ist.psu.edu/diffie76new.html>. Acesso em: 15 mar. 2005.

Stallings, W. Network Security Essentials: applications and standards. New Jersey: Prentice Hall, 1999.

Menezes, A. J.; Oorschot, P. C. V.; Vanstone S. A. Handbook of Applied Cryptography, Out., 1996.

Wolfgang, P. Design patterns for object-oriented software development. 2rd edition. 1995.

RFC 2440 (1998) “Open PGP Message Format ”, J. Callas, L. Donnerhacke, H. Finney, R. Thayer, Nov. 1998.

Lehtonen, S.; Pärssinen, J. Pattern language for Cryptographic Key Management. EuroPLoP, 2002.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns. Addison Wesley, Reading, MA, 1995.

Watkins, D. Overview and Comparison of GSM, GPRS and UMTS. Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, abr. 2000.

Biryukov, A.; Kushilevitz, E. From Differential Cryptanalysis to Ciphertext-Only Attacks. In: CRYPTO, pp72–88, 1998.

deSouza, J. T.; Matwin, S. A Pattern Language for Providing Client-Server Confidential Communication, in: SugarLoafPLop, Rio de Janeiro, Brazil, 2001.

S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In Eighth Annual Workshop on Selected Areas in Cryptography, Toronto, Canada, Aug. 2001.

Patterns for Parallel and Distributed Processing of Large Hierarchical Structures

Denise Stringhini Ismar Frango Silveira Luciano Silva
Faculdade de Computação e Informática, Universidade Presbiteriana Mackenzie
{dstring, ismar, lucianosilva}@mackenzie.br

Abstract. *Processing of large hierarchical structures could achieve better performance whether implemented over parallel environments. Although there is a wide range of applications for such parallelized structures, there is a lack of well-defined design patterns to model them, even though some parallel programming patterns have already been proposed. This paper proposes a set of design patterns that address these issues as well as describing some potential applications.*

1 Introduction

Parallelism has been largely used to obtain better performance in computational intensive applications. Nowadays, clusters of thousands of processors are used to run scientific applications such as weather forecasting or genomic sequencing processing. Parallel processing is also used in industry for applications that range from web search to computer graphics. This processing environment is largely improved by the use of cluster computing technology. Programming in such environments is usually enhanced by parallel programming libraries such as MPI [Gropp *et al*. 1999] and PVM [Geist *et al*. 1994]. Also, Grid environments take advantage of the Internet infrastructure to allow the execution of distributed applications over geographically separated machines.

Despite the increasing amount of research on techniques and tools for parallel and distributed programming, this is still a hard task considering distributed memory architectures. Issues like data and task partitioning, data mapping, communication and synchronization between processes are difficult to manage. Besides, these issues must be addressed to achieve the best possible performance.

Recently, object-oriented design patterns have been receiving more attention from parallel programming designers. This is probably because it is possible to identify patterns in some parallel programming techniques such as the *bag of tasks*, *pipeline*, *divide and conquer*, *master/workers* [Mattson *et al*. 2004]. The main advantage of using patterns is their independency on languages, libraries, or tools. In the context of parallel and distributed programming this is particularly useful since the available hardware and tools could vary considerably.

This paper presents a set of design patterns for parallel and distributed processing of large hierarchical structures, such as scene graphs and phylogenetic trees, distributed over a cluster of computers. Section 2 describes some related work. Section 3 presents the proposed patterns. Section 4 discusses some patterns applications and finally some conclusions and future works are presented in Section 5.

2 Related work

Frameworks and skeletons have been developed to help parallel and distributed programming. For example, there is the PAS – Parallel Architectural Skeleton and *SuperPAS* [Akon *et al.* 2004] system that is a pattern-based parallel programming model and environment. *SuperPAS* is an extension of PAS that provides a skeleton description language for the generic PAS. In this approach the user has to learn the system's description language, which is sometimes undesirable.

There is some interesting work in developing parallel design patterns. Mattson [Mattson *et al.* 2004] presents a collection of patterns for parallel programming based on classical patterns description. They describe a pattern language organized into four design spaces: finding concurrency, algorithm structure, supporting structures, and implementation mechanisms. The programmer has to consider each of these spaces in order to complete an application. Each space is composed of a collection of parallel design patterns. Nonetheless, this approach doesn't address conceptual design issues since it is limited to low-level coding.

The CO_2P_3S – Correct Object-Oriented Pattern-based Parallel Programming System [Tan *et al.* 2003] and MetaCO_2P_3S, use generative design patterns. A programmer selects the parallel design patterns and then generates a custom framework for the application that includes all structural code necessary for the application to run in parallel. The programmer is only required to write simple code that launches the application and to fill in some application-specific sequential hook routines. These tools don't substitute pattern definitions since they are required by MetaCO_2P_3S to provide basic information in order to allow CO_2P_3S to generate code.

3 Proposed Patterns

In order to contribute to the definition of patterns for parallel and distributed applications, three patterns are being proposed as follows: Distributed Composite, Dual Visitor and Matched Transporter.

3.1 Distributed Composite

Motivation: Parallel applications often need to work with data that are organized in a hierarchical structure, making each one of the components highly dependent on their ancestors or descendants, according to the traversal strategy. Such dependence makes the parallel processing of these elements more difficult, since the nodes have to be distributed over several processors.

Intent: Distributed Composite lets clients treat individual objects and compositions of objects uniformly, whether remote or local.

Applicability: The use of the Distributed Composite pattern is recommended when:

- Distributed objects are organized in a large hierarchical, tree-like structure;
- Their processing can be done in parallel, usually by bottom-up traversal strategies.

Structure: The key component in Distributed Composite is an abstract class *Node* that is implemented by classes that represent local nodes and remote references. Local nodes

may represent primitive elements or composite nodes, as in the Composite pattern [Gamma *et al.* 1995]. The structure for the Distributed Composite Pattern is shown in Figure 1.

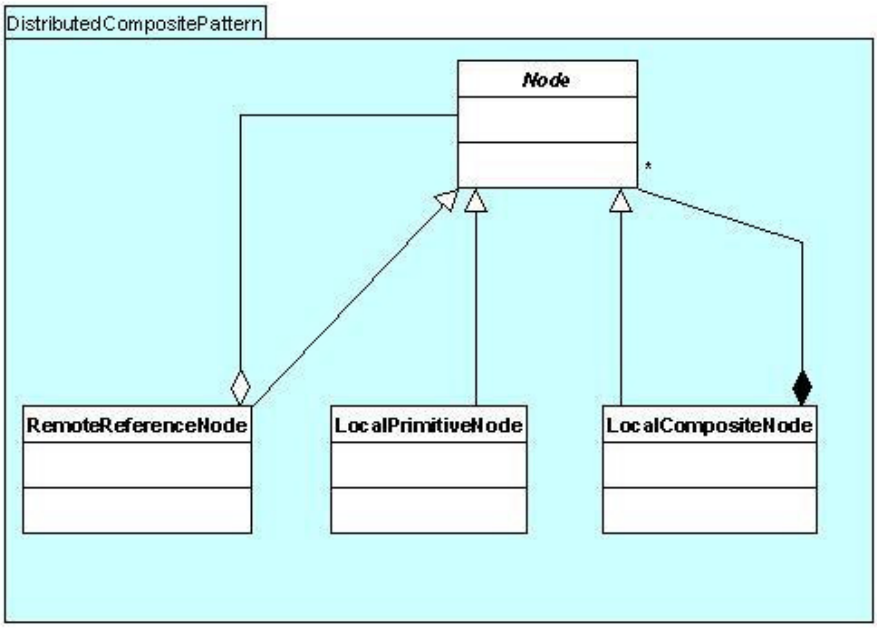


Figure 1. Distributed Composite Pattern

Participants:

- **Node**: this abstract class implements default behavior for the interface common to all classes. It defines an interface for accessing and managing its child components, and for accessing a parent component in the recursive structure (these methods are not shown in model, since they are already defined in the Composite pattern [Gamma et al. 1995]). Since this class represent nodes that will be processed by Dual Visitor, it must include an accept( ) method, which must be implemented by their subclasses.
- **LocalPrimitiveNode**: represents a leaf object that will be locally processed.
- **LocalCompositeNode**: represents nodes with children, which can be of any type that implements *Node*.
- **RemoteReferenceNode**: instances of this class actually point to an instance of any subclass of *Node* that is not in the same physical or logical context.

Collaborations:

- Clients use Node interface to interact with different objects in the structure. If the object is an instance of LocalPrimitiveNode requests can be treated directly. If it is a LocalCompositeNode, requests are recursively forwarded to the next nodes in hierarchy. However, if it is a RemoteReferenceNode requests are sent to remote nodes pointed by this object. These requests are treated as local ones by the remote nodes.

Consequences:
The Distributed Composite pattern:

- defines class hierarchies consisting of local objects and remote objects. Wherever client expects a local object, it can take a remote object. This remote object represents the remote processing of a subtree. It considers the previous partitioning of the hierarchical application (the tree) that could be distributed across several processors.
- make clients simpler, since they could treat composite structures, local objects and remote objects in the same way.

Sample Code: the following is an example of implementation in Java, using RMI (*Remote Method Invocation*) as basis for implementing remote nodes.

```
import java.rmi.*;
import java.rmi.server.*;

interface Node
   extends Remote
{
 //methods
};

class LocalPrimitiveNode
   extends UnicastRemoteObject
   implements Node
{
 // problem-specific
 // implementation
}
```

```
class CompositeNode
     extends UnicastRemoteObject
     implements Node
{
 private Node composite[];
 // problem-specific
 // implementation
}

class RemoteReferenceNode
    implements Node
{
 private Node remoteReference;
}
```

Known uses:

- Johnson and Krishna's (1993) early works deal with distributed B-trees (dB-trees) to illustrate techniques for designing distributed search structures.
- Cluster-wide JNDI (Java Naming and Directory Interface) trees are similar to a single server instance JNDI tree. In addition to storing the names of local services, however, the cluster-wide JNDI tree stores the services offered by clustered objects from other server instances in the cluster. Application servers like WebLogic (Prem *et al*., 2003) and JBoss (Burke and Labourey, 2002) support this kind of JNDI implementation.
- Brushwood (Zhang *et al*., 2005) uses a distributed implementation of a B-tree to provide a framework for implementation of p2p applications.
- Yilmaz and Erdoğan (2001) present a model called Distributed Composite Object (DCO). They have also designed and implemented a software layer, DCOBE (Distributed Composite Object Based Environment) that can be placed on top of Java programming language to provide a uniform interface for collaborative application developers to use.

Related patterns: the nodes in the Distributed Composite structure will accept a Dual Visitor that will change the node strategy depending on the type of the Node. If the recipient is a Remote Reference Node, the Matched Transporter pattern will be used to

transport the Dual Visitor to the matched Remote Reference Node (probably in a remote processing unit). Once the transported Dual Visitor reaches its remote counterpart, it triggers the remote processing of another Distributed Composite structure, while the original structure could still be visited.

3.2 Dual Visitor

Motivation: Processing heterogeneous hierarchical structures frequently demand different processing strategies that depend on the classes the objects belong to. This problem can be solved with the Visitor pattern [Gamma *et al.* 1995]. In distributed applications whose data is structured according to the Distributed Composite pattern, all remote references must be handled in such a way that the processing could be parallelized without generating deadlocks or other faults.

With the Dual Visitor pattern, which is an extension of Visitor and Strategy patterns [Gamma *et al.* 1995], different visiting strategies – for local and remote elements – may be dynamically exchanged, according to the type of object being visited. Thus, different remote visiting strategies may be implemented according to system requirements.

Intent: Dual Visitor allows interchangeable visiting strategies tailored to the kind of objects being visited. This is done without changing the classes of the elements on which it operates, or the Visitor object itself.

Applicability: Dual Visitors should be used in the following situations:
- The hierarchical structure to be visited is distributed in such a way that all parts are well-connected by remote references;
- Each part of the tree can be independently visited.

Structure: The structure for the Dual Visitor Pattern can be seen in Figure 2.

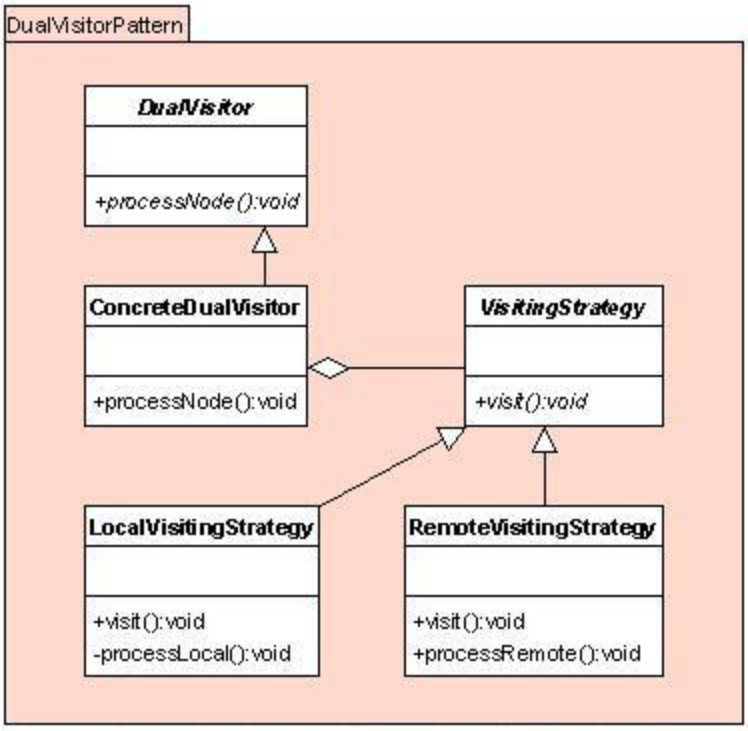


Figure 2. Dual Visitor Pattern

Participants:

- **DualVisitor**: abstract class that defines a *processNode( )* operation that must be implemented by its subclasses. Its use is recommended for extensibility reasons.
- **ConcreteDualVisitor**: implements the *processNode( )* operation by calling the *visit()* operation defined in *VisitingStrategy*.
- **VisitingStrategy**: defines an abstract *visit* operation that will be implemented by its subclasses. Using this class, it is possible to change its implementation without changing the instance of *ConcreteDualVisitor* in a tree segment.
- **LocalVisitingStrategy**: contains visiting methods for local nodes. These methods are called by the implementation of *visit( )*.
- **RemoteVisitingStrategy**: defines a strategy for processing remote nodes.

Collaborations: when a node accepts a Dual Visitor it could either continue local processing or trigger remote parallel processing (if it is a Remote Reference Node). Figure 3 illustrates the collaborations between the Dual Visitor, the local and remote strategies and the two types of nodes (local and remote).

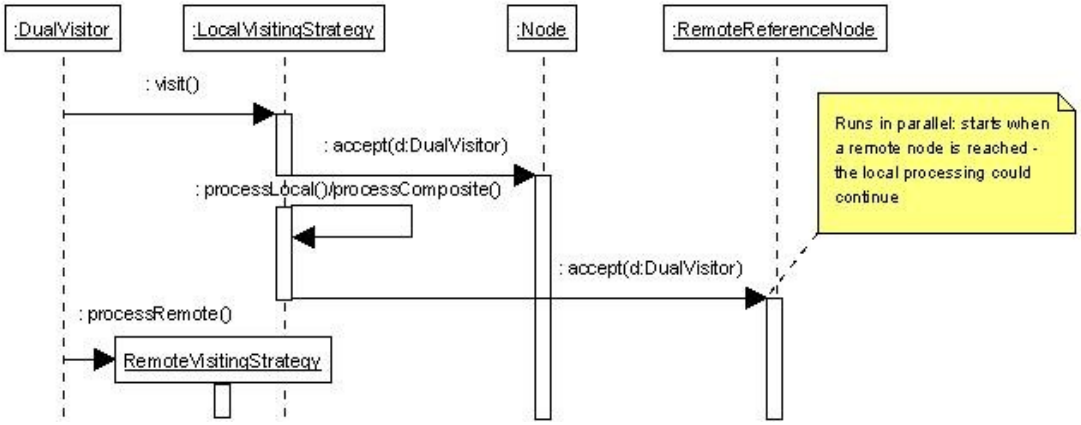


Figure 3. UML Sequence diagram for the Dual Visitor Pattern

Consequences:

The Dual Visitor pattern:

- makes transparent the visiting strategy being used. Clients do not need to know if the structure being visited is local or remote. These strategies are easily interchanged.
- allows the accumulation of states, whether local or remote, during the visiting process of a previously distributed hierarchical structure.
- solves a common problem found in Visitor pattern (Gamma, 1995): it is hard to add different elements to be visited once a Visitor is already defined. Dual Visitor solves this by applying Strategy pattern (Gamma, 1995) to decouple visiting strategies from Dual Visitor's structure.

Sample code: the following is a Java code with an implementation for Dual Visitor Pattern.

```
/* Node class stands for any tree element */
interface DualVisitor
{
   void processNode(Node n);
}

class ConcreteDualVisitor
   implements DualVisitor
{
   private VisitingStrategy vsBridge;

   void setStrategy(Node n)
   { n = /* n is Local*/ ? LocalVisitingStrategy.getInstance()
                         : RemoteVisitingStrategy.getInstance();
   }
   void processNode(Node n)
   {
        this.setStrategy(n);
        vsBridge.visit(n);
   }
}

interface VisitingStrategy
{
   void visit(Node n);

}

class ConcreteVisitingStrategy
   implements VisitingStrategy
{
   void visit(Node n)
      {
     this.processLocal(n);
   }

   void processLocal(Node n)
   {
    //Process problem-specific business rules
   }
}

class RemoteVisitingStrategy
   implements VisitingStrategy
{
   void visit(Node n)
   {
     this.processRemote(n);
   }

   void processRemote(Node *n)
   {
      /*Visiting strategies in some part of the tree hosted by
     another machine are triggered. Locally, processing
     can continue in parallel - or wait, if needed.*/
   }

}
```

Known uses:

- Martin (1997) proposes a variation of Visitor pattern (Gamma, 1995) called Acyclic Visitor, which allow new functions to be added to existing class hierarchies without affecting those hierarchies, and without creating some dependency cycles that are inherent to Visitor.
- Adaptive programming (Yoder and Razavi, 2000) allows capturing crosscutting concerns by structure-shy adaptive visitors. Demeter Tools (DRG, 2000), use extensively a Selective Visitor in order to loosely couple behavior modification to behavior and structure.
- DJ library (Orleans and Lieberherr, 2001) is an aspect-oriented (Filman et al., 2005) Java library for adaptive programming that allows traversal strategies to be constructed and interpreted dynamically at run-time by reflection-based Adaptive Visitors.
- JAsCo (Suvée et al., 2003) is an aspect-oriented programming language targeted at Component-Based Software Development. Vanderperren et al. (2005) present an implementation of an adaptive visitor as a regular JAsCo aspect bean.

Related patterns:

- Dual Visitors are well-suited to traverse Distributed Composite structures.
- Interpreters (Gamma, 1995) could be attached to Dual Visitors in order to perform interpretation according to the behavioral aspects of the structure being visited.

3.3 Matched Transporter

Motivation: When a remote reference is reached, an object is needed to trigger the change of visiting strategy. The object must remotely activate new processing tasks or swap remote references and tree segments. The Matched Transporter pattern combines Observer flexibility [Gamma *et al.* 1995] and the Data Transfer Object facilities [Alur *et al*. 2003]. A local observer is attached to every remote reference at the same time it is matched to another remote observer. State changes in one observer will be immediately reflected in the remote counterpart. This behavior resembles the EPR pairs of Quantum Mechanics [Griffith 2004].

Intent: This pattern defines a dependency among local objects and a mapping between remote objects. When one object state changes, all its local dependents are notified and updated just in time and its remote counterpart assumes its state.

Applicability: Matched Transporters are useful in the following situations:

- While traversing a hierarchical structure, Dual Visitors need to be warned when remote nodes are reached in order to exchange their visiting strategies;
- Objects must be transferred after completing of local processing to replace remote references to them.

Structure: Figure 4 shows the general structure for the Matched Transporter pattern.

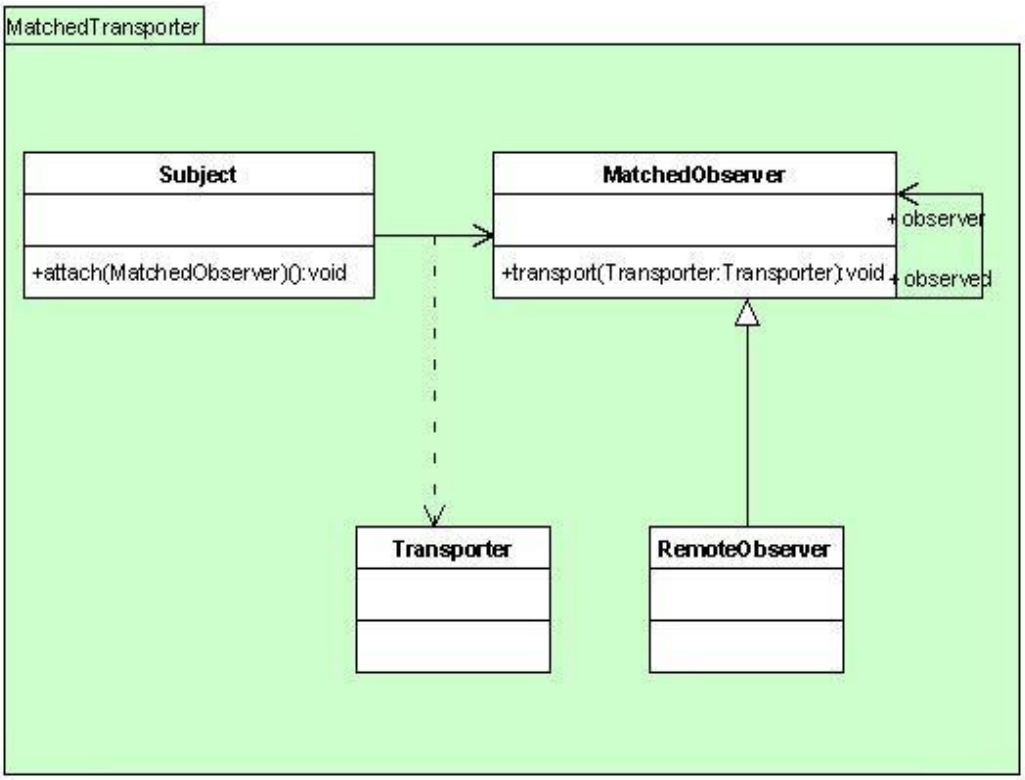


Figure 4. Matched Transporter Pattern

Participants:

- **MatchedObserver**: concrete class that maintains a recursive association. Each instance will remotely refer to a shadow sibling and both are self reflections.
- **Transporter**: associative class between *MatchedObservers*, responsible for the transfer of *Nodes*.
- **RemoteObserver**: subclass of *MatchedObserver* locally detects objects of Distributed Composite that are instances of *RemoteReferenceNode*, in order to warn the local instance of *DualVisitor* to change its implementation for *VisitingStrategy*.

Consequences:

The Matched Transporter pattern:

- supports instant peer-to-peer communication through Matched Observers. Since Matched Observers are entangled, any event observed is immediately reported to both peers.
- relies on object persistence mechanisms. Such condition could lead to implementations that are language or platform-dependent.

Sample Code: a possible implementation of Matched Transporter in Java follows, using RMI and object serialization to implement this pattern.

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

class Transporter
    implements Serializable, Remote
{
    private Serializable wrapped;

    public Transporter (Serializable s)
    {
      wrapped = s;
    }
}

interface IMatchedObserver
     extends Remote
{
    public void attach (IMatchedObserver mob, Transporter t);
}

class MatchedObserver
    extends UnicastRemoteObject
    implements IMatchedObserver
{
 private OutputStream entanglementOut;
 private InputStream entanglementIn;
 private IMatchedObserver eprPeer;
 private Transporter tRef;

 /*All attributes with protected getters and setters*/
 public MatchedObserver ()
 {
 /*Entanglement attributes must be properly initiated here*/
 }

 public void attach (MatchedObserver mob, Transporter t)
 {
   tRef=t;
   if (eprPeer==null)
   { eprPeer = mob;
     eprPeer.attach(this,tRef);
   }
 }

 public boolean catch()
    throws IOException, ClassNotFoundException
 {
   tRef = (Transporter) entanglementIn.readObject();
 }

 public void transport (Transporter t) throws IOException
 {
   entanglementOut.writeObject(t);
   eprPeer.catch();
   entanglementOut.close();
 }
}
```

```
class RemoteObserver
     extends MatchedObserver
{
   public RemoteObserver () throws RemoteException
   {
   /*Remote entanglement must be configured here*/
   }
   public void attach (IMatchedObserver mob, Transporter t)
                  throws RemoteException
   {
   // Ovverides this method, using RMI to get entangled
      // to a remote peer
   }
}

class Subject
   implements Serializable
{
   transient private MatchedObserver mob;
   //problem-specific non-transient attributes
   public Transporter attach (MatchedObserver mob)
   {
    Transporter t=new Transporter(this);
       this.mob = mob;
       mob.attach(this, t);
       return t;
      }
      //getters and setters
}
```

An alternative implementation could consider *MatchedObserver* as a specialization of *Subject*, from the point of view of its peer observer.

Collaborations: the Matched Transporter transfers the context of a *Node* between two matched *RemoteObservers*. The following sequence diagram (Figure 5) illustrates the collaborations between local and remote elements.

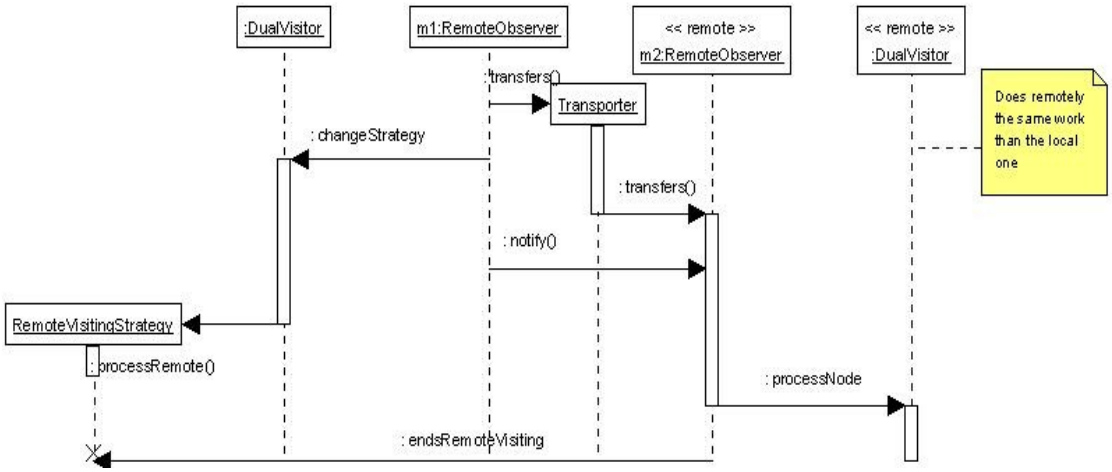


Figure 5. Collaboration diagram for the Matched Transporter Pattern

Known Uses:

- Halbwachs et al. (1993) present a formal specification of synchronous observers for reactive systems. This kind of observer is used, for instance, in Xeve (Bouali, 1998), which is a graphical interface environment for symbolic analysis and verification of Esterel programs modeled as Finite State Machines.
- Rieffel and Polak (2000) present the teleportation problem in Quantum Cryptography: the objective is to transmit the quantum state of a particle using classical bits and reconstruct the exact quantum state at the receiver. Both transmitter and receiver act as MatchedObservers and the Transporter is the entanglement itself.
- Harrison, Levine and Schmidt (1997) propose a Real-Time Event Service for TAO, a Real-Time CORBA architecture. The service allows CORBA Event Channel to support synchronous Real-Time event dispatching. Under this mechanism CORBA Event Channel acts as a RemoteObserver of suppliers' Events and clients' Requests (the Transporters).

Related patterns:

- Matched Transporters could be used to transfer Distributed Composite structures between different nodes.
- A Mediator (Gamma, 1995) could be attached to Matched Transporter to allow multicast communication.

4 Patterns Applications

In this section we present some problems whose static and dynamic structures could be treated with our patterns. We choose hard-processing and wide area applications in order to demonstrate the effective use of the proposed patterns.

4.1 Scene Graph Distributed Rendering

A *scene graph* is a hierarchical structure built from *nodes*. It is a common data structure which arises in several areas such as computer graphics [Lengyel 2003], image processing [Nokolaids and Pitas 2000], computer vision [Forsyt and Ponce 2003] and virtual reality [Bowman *et al* 2004].

Each scene graph node encapsulates some characteristic related to scene description:

- object geometric parameters (geometric type, radius, size, position)
- transformations (translation, rotation, scaling)
- appearance information for rendering (color, texture, reflection parameters)
- behaviors
- visualization and environment setup

The scene graph transformation onto images is called *rendering* and usually requires a powerful processing environment. Some important libraries like Java3D, OpenInventor and OpenSceneGraph use scene graphs extensively as the main representation of their graphical cores.

Figure 6 shows a typical Java3D scene graph structure:

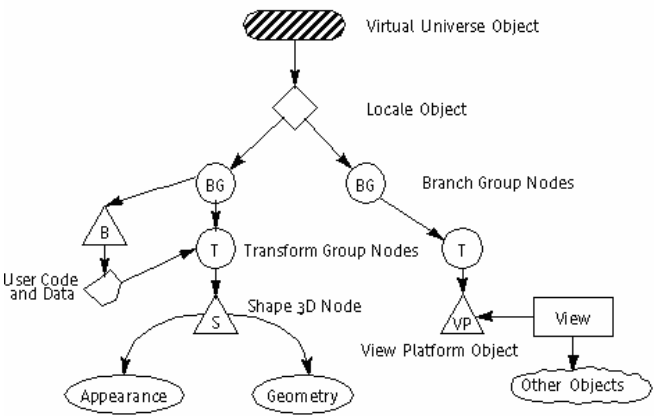


Figure 6. A typical Java3D Scene Graph Structure

There are two natural scene graph decompositions: transform nodes and branching groups. Figure 7 shows an example of scene graph decomposition using the patterns proposed in this paper.

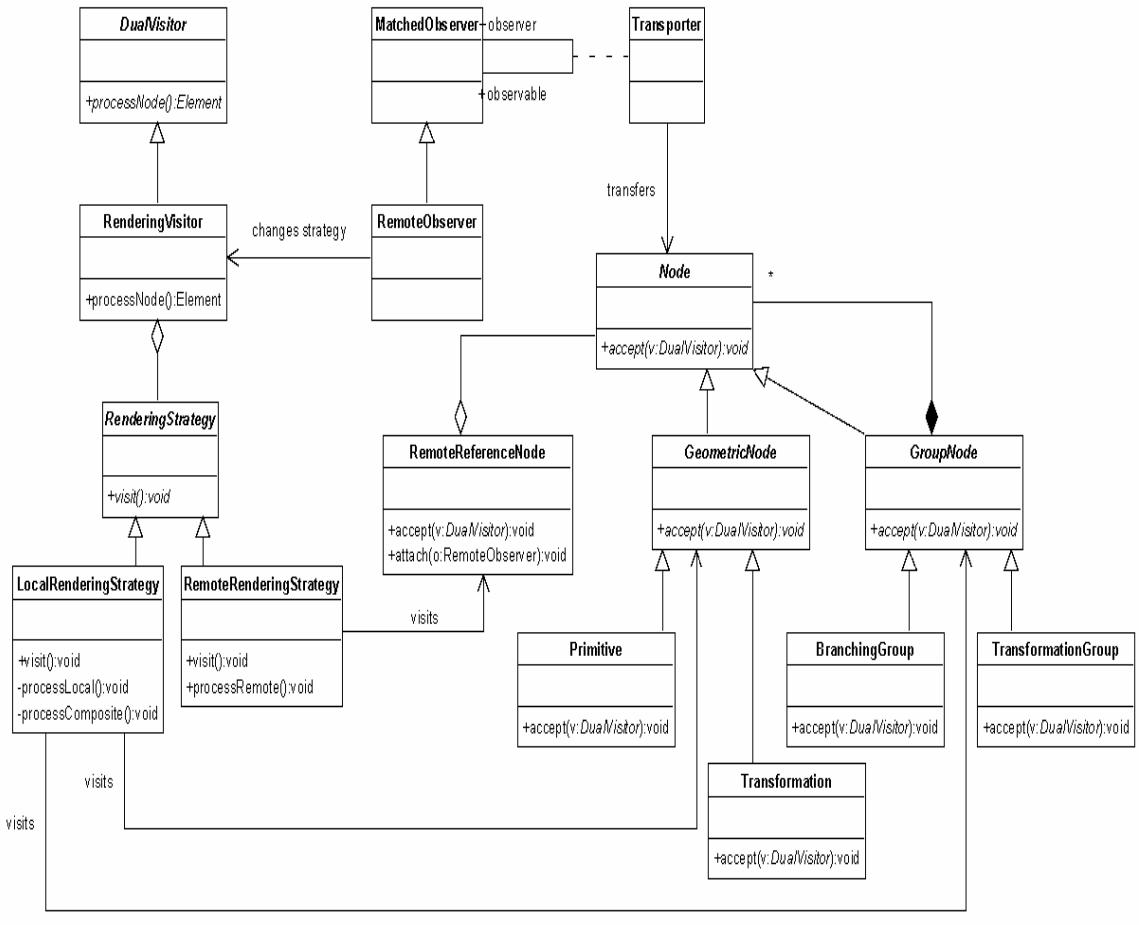


Figure 7. Scene graph decomposition into local and remote nodes

Transform nodes affect groups of primitives and usually are stored in the same host. Branching groups are sets compound by objects and transformations. There is at most

one RemoteReferenceNode for each Node. This allows the construction of Node pairs: one of them is a tree root and another is a leaf. In Figure 7, *GeometricNode* and *GroupNode* represent local information. Several rendering strategies may be used for scene graph transversal and are locally implemented by *processLocal()* and *processComposite()* methods in the *LocalRenderingStrategies*. The proposed pattern allows arbitrary scene graph decomposition, which provides great flexibility in achieving improved processing performance.

4.2 Phylogenetic Trees and Cladograms

A *phylogenetic tree* [Pevzner 2000] is a graphical representation of the evolutionary relationship between taxonomic groups, as depicted on the Figure 8. The term phylogeny refers to the evolution or historical development of a plant or animal species, or even a human tribe or similar group.

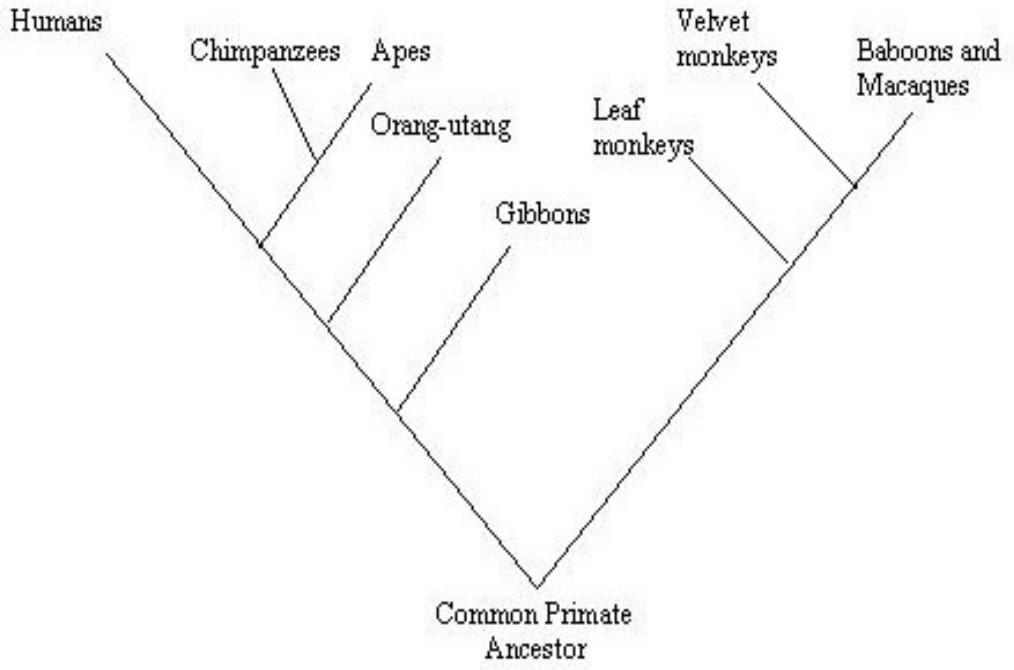


Figure 8. Cladogram of human race and its relationships with another species

A phylogenetic tree is a specific type of *cladogram* where the branch lengths are proportional to the predicted or hypothetical evolutionary time between organisms or sequences. The cladogram only illustrates the probability that two organisms, or sequences, are more closely related to each other than to a third organism, it does not necessarily clarify the pathway that created the existing relationships. However, the cladogram can be used in the formulation of new hypotheses and to cast new light on existing data. For this formulation, several operations over cladograms are needed in order to obtain predictions about the structure. Even in supercomputers or clusters, the cladogram analysis depends on high efficient data structures and algorithms.

Both phylogenetic trees and cladograms are compound structures (trees). Leaves represent elements called *taxons* and internal branching nodes can contain a wide range

of values. It is easy to see that the DistributedComposite pattern could be easily adapted to represent those structures, as shown in Figure 9.

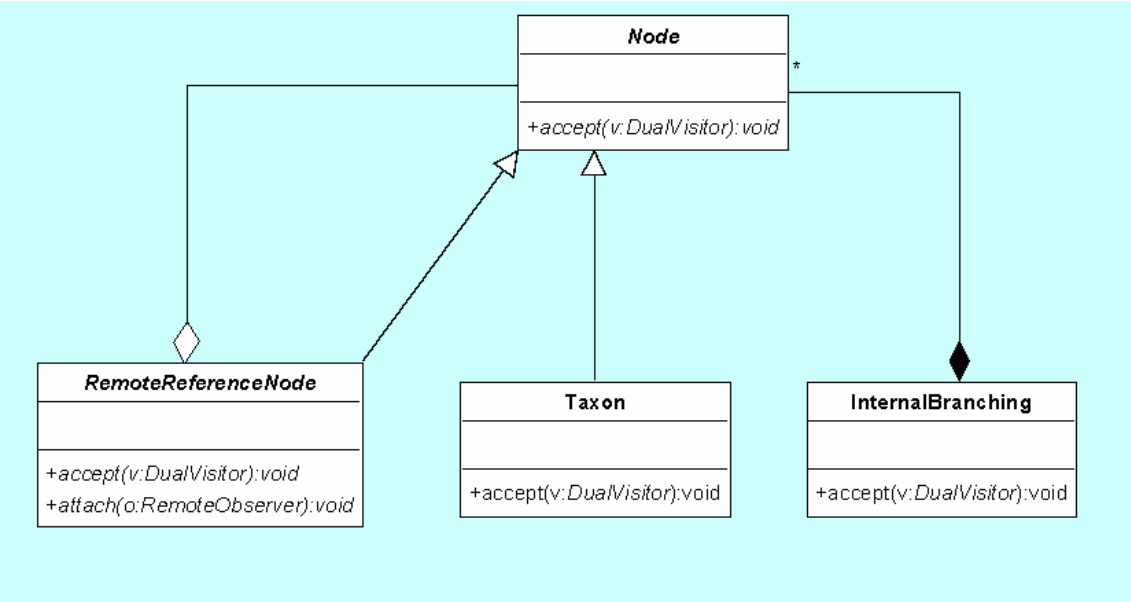


Figure 9. Cladogram decomposition into local and remote components

Several algorithms which traverse phylogenetic and cladograms, like parsimony or distance algorithms, may be now embedded into DualVisitor objects. By changing the DualVisitor's strategy one can use different transverse algorithms with minimal impact on phylogenetic or cladograms data structures.

5 Conclusions and further work

Parallel processing of large hierarchical structures includes a wide range of applications. There is a lack of design patterns to model them. Since distribution and processing of these structures are non-trivial, hard computational tasks, the development of adequate design patterns could improve the reusability and expansibility of common solutions in this context.

This paper presented a set of design patterns for parallel processing of large hierarchical structures, which could be distributed over a cluster of computers. These patterns incorporate well-known, classical patterns, which guarantee high cohesion and low coupling between classes. The main contribution was to provide a generic solution for modeling and traversing distributed tree-like structures, which arise naturally in many applications. The proposed patterns allow different implementations of user-defined processing and traversal strategies.

Future work will include the implementation of these patterns over different parallel and distributed architectures and applications. Tree partitioning optimal strategies are also a candidate for future research.

The authors would like to acknowledge Cleber Ferreira de Castro Marchetto Zarate for his work on initial implementation of these patterns in MPI and RMI, which contributed to provide sample codes for patterns' descriptions.

References

Akon, M. M. *et al*. (2004), "SuperPAS: A Parallel Architectural Skeleton Model Supporting Extensibility and Skeleton Composition". In Proceedings: *Second International Symposium on Parallel and Distributed Processing and Applications* (ISPA'04), Hong Kong, p. 985-996.

Alur, D. *et al*. (2003), *Core J2EE Patterns*, 2nd Edition, Prentice Hall.

Gamma E. *et al*. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Bouali, A. (1998) "XEVE, an ESTEREL Verification Environment". *Lecture Notes In Computer Science*; v. 1427.

Bowman, D.A., Kruijff, E., LaViola, J.J. and Poupyrev, K. (2004), I. 3D User Interfaces: Theory and Practice. Addison-Wesley Professional.

Burke, B. and Labourey, S. (2005), "Clustering with JBoss 3.0" In: *OnJava electronic magazine*. Available on the Internet at http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html (Visited september, 5, 2005).

Filman, R.; Elrad, T.; Clarke, S. and Askit, M. (2005) *Aspect-Oriented Software Development*. Addison-Wesley.

Forsyt, D.A. and Ponce, J. (2003), Computer Vision: A Modern Approach, Prentice Hall.

Geist, M. et al. (1994), *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Parallel Computing*. MIT Press.

Griffiths, D.J. (2004), *Introduction to Quantum Mechanics*, Prentice Hall, 2nd Edition.

Gropp, W. et al. (1999), *Using MPI: Portable Parallel Programming with the Message Passing Interface* (Scientific and Engineering Computation Series). MIT Press.

Halbwachs, N.; Lagnier, F. and Raymond, P. (1993) "Synchronous Observers and the Verification of Reactive Systems". *Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology*. London: Springer-Verlag.

Harrison H. T., Levine D. L. and Schmidt D. C., (1997) "The Design and Performance of a Real-time CORBA Event Service," in Proceedings of OOPSLA '97, Atlanta, GA, October, ACM.

Johnson, T. and Krishna, P. (1993) "Designing Distributed Search Structures with Lazy Updates". Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. Washington.

Lengyel, E. (2003), Mathematics for 3D Game Programming and Computer Graphics, Charles Rivers Media, 2nd edition.

Martin, R. C. (1997) Pattern languages of program design 3. Addison-Wesley.

Mattson, T. G. et al. (2004), Patterns for Parallel Programming. Addison-Wesley.

Nikolaidis, N. and Pitas, I. (2000), 3-D Image Processing.Willey Interscience.

Orleans, D. and Lieberherr, K. (2001) "DJ: Dynamic Adaptive Programming in Java". In: Proceedings of Reflection 2001 - The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. Kyoto, Japan.

Pevzner, P.A. (2000), *Computational Molecular Biology: An Algorithmic Approach*. MIT Press.

Prem, J.; Ciconte, B.; Devgan, M.; Dunbar, S. and Go, P. BEA (2003) *WebLogic Platform 7*. Sams Publishing.

Rieffel, E. and Polak, W. (2000) "An introduction to quantum computing for non-physicists". *ACM Computing Surveys* v. 32 (3), pp. 300-335, September, New York: ACM Press

Suvée, D.; Vanderperren, W. and Jockers, V. (2003) "JAsCo: an aspect-oriented approach tailored for component based software development". *Proceedings of the 2nd international conference on Aspect-oriented software development*. Boston

Tan, K. et al. (2003), "Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment". In Proceedings: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'2003)*.

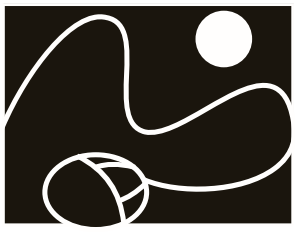
The Demeter Research Group. (2005) Online Material on Adaptive Programming and Demeter/Java. Available on the Internet at http://www.ccs.neu.edu/research/demeter/, 2000. Visited at September, 4

Vanderperren, W.; Suvée, D.; Verheecke, B.; Cibrán, M. A.; Jonckers, V. (2005) "Adaptive programming in JasCo". In: *Proceedings of the 4th international conference on Aspect-oriented software development*. Chicago.

Yilmaz, G. and Erdoğan, N. (2001) "A New Distributed Composite Object Model For Collaborative Computing". *Proceeding of ISCIS 2001 - International Symposium on Computer and Information Sciences*. Antalya, Turkey, November.

Yoder, J. W. and Razavi, R. (2000) "Metadata and adaptive object-models". *In ECOOP 2000 Workshop Reader, volume 1964 of Lecture Notes in Computer Science*. Springer Verlag

Zhang, C.; Krishnamurthy, A.; Wang, R.Y. (2005) "Brushwood: Distributed Trees in Peer-to-Peer Systems". *Proceedings of IPTPS'05 – International Workshop on Peer-To-Peer Systems*. Ithaca, New York, February.



SugarLoaf PLoP'2005

Pattern Applications

Padrões e Métodos Ágeis: agilidade no processo de desenvolvimento de software

Edes Garcia da Costa Filho[1,*] , Rosângela Penteado[1]

Júnia Coutinho Anacleto Silva[1], Rosana Teresinha Vaccare Braga[2]

[1]Universidade Federal de São Carlos – Departamento de Computação

[2]Universidade de São Paulo – Instituto de Ciências Matemáticas e de Computação

{edes_filho,rosangel,junia}@dc.ufscar.br, rtvb@icmc.usp.br

Abstract. *This paper presents some organizational and process patterns that can be integrated to agile methods to improve and speed up the software development process. Common features can be found in some organizational and process patterns, as well as in practices used in agile methods such as Extreme Programming (XP) and Scrum. From these features, some patterns that are used as practices in these methods and others that can be integrated to them were emphasized. There is still the challenge to integrate these patterns to an agile method.*

Keywords: Organizational and Process Patterns, Agile Methods, Extreme Programming, Scrum.

Resumo. *Este artigo apresenta alguns padrões organizacionais e de processo que podem ser integrados aos métodos ágeis para melhorar e agilizar o processo de desenvolvimento de software. Características comuns podem ser encontradas em alguns padrões organizacionais e de processo existentes e em algumas práticas utilizadas em métodos ágeis como o Extreme Programming (XP) e o Scrum. A partir dessas características, foram destacados alguns padrões que são usados como prática nesses métodos e outros que podem ser integrados a eles. Existe ainda o desafio de integrar esses padrões a um método ágil.*

Palavras chave: Padrões Organizacionais e de Processo, Métodos Ágeis, Extreme Programming, Scrum.

1. Introdução

Um desafio constante da área de Engenharia de Software é melhorar o processo de desenvolvimento de software. Mesmo com a constante evolução de métodos, técnicas e ferramentas, a entrega de software em prazos e custos estabelecidos nem sempre é

* Apoio Financeiro do CNPq

conseguida. Uma das causas desse problema é o excesso de formalidade nos modelos de processo propostos nos últimos 30 anos (Fowler, 2003; Larman, 2004). Existe hoje a necessidade de desenvolver software de forma mais rápida, mas com qualidade. Esse desenvolvimento pode ser obtido utilizando métodos ágeis e padrões organizacionais e de processo. A popularização dos métodos ágeis ocorreu com "Manifesto Ágil" (Beck et al., 2001), que indica alguns princípios que são compartilhados por tais métodos:

- Indivíduos e interações são mais importantes que processos e ferramentas;
- Software funcionando é mais importante do que documentação detalhada;
- Colaboração dos clientes é mais importante do que negociação de contratos;
- Adaptação às mudanças é mais importante do que seguir um plano.

Nos últimos anos, métodos ágeis como o XP (Beck, 1999), Scrum (Schwaber, 2002) e Crystal (Cockburn, 2002) passaram a ser usados em empresas, universidades, institutos de pesquisa e agências governamentais (Goldman et al., 2004).

O reuso de software é uma atividade comum durante o processo de desenvolvimento. Juntamente com outras técnicas, por exemplo desenvolvimento de software baseado em componentes, os padrões de software (software *patterns*) auxiliam e contribuem para o reuso em níveis mais altos de abstração, como por exemplo, em nível de análise, arquitetural, organizacional e de processo. Das diversas categorias de padrões que surgiram, os padrões organizacionais e de processo são os que têm por objetivo apoiar a construção do software e melhorar o seu desenvolvimento. Além de estarem divididos em categorias, os padrões podem ser agrupados em linguagens de padrões. Uma linguagem de padrões é um sistema de padrões organizados em uma estrutura que guia a sua aplicação (Kerth et al., 1997).

Os padrões organizacionais e de processo cobrem problemas de desenvolvimento. Eles podem ser usados para modelar uma nova organização e seu processo de desenvolvimento (Coplien, 1995). Essas duas categorias de padrões podem ser utilizadas em conjunto com métodos ágeis.

A simples utilização de métodos ágeis pode não suprir as necessidades de uma organização ou projeto. Adaptações nos métodos podem ser necessárias (Goldman et al., 2004; Taber et al., 2000). Os padrões organizacionais e de processo podem ser utilizados nesse contexto para ajustar ou aperfeiçoar um método ágil, como o XP ou Scrum, de acordo com as necessidades da organização ou do projeto.

Neste artigo, são apresentados dez padrões organizacionais e de processo, propostos por diferentes autores, que podem ser integrados aos métodos ágeis, para melhorar ou adaptar o método ágil e dar mais agilidade ao processo de desenvolvimento. Essa integração será avaliada por meio de estudos de caso. Na Seção 2, são apresentados alguns trabalhos relacionados encontrados na literatura. Na Seção 3, são apresentados conceitos básicos sobre os métodos ágeis XP e Scrum. Na Seção 4, são apresentados os padrões organizacionais e de processo que já são encontrados nos métodos ágeis e os que podem ser integrados a eles. Finalmente, na Seção 5, estão as considerações finais.

2. Trabalhos Relacionados

Algumas publicações encontradas na literatura apresentam padrões organizacionais e de processo para desenvolvimento ágil de software e a integração de padrões organizacionais e de processo com métodos ágeis.

A linguagem de padrões "*A Generative Development-Process Pattern Language*" (Coplien, 1995), publicada em *Pattern Languages of Program Design* (Coplien et al., 1995) é a primeira referência sobre padrões organizacionais e de processo. Essa linguagem possui um conjunto de padrões organizacionais e de processo de sucesso, que podem ser utilizados para estabelecer estruturas organizacionais e práticas cujo objetivo é melhorar o processo de desenvolvimento de software da organização.

Coplien et al. (2004) apresentam quatro linguagens de padrões que combinam estruturas organizacionais com as melhores práticas de desenvolvimento de software. Essas linguagens, que devem ser usadas em conjunto para solucionar problemas de desenvolvimento de software da organização, são:

- *Project Management Pattern Language*: trata do trabalho e estruturação da organização, com foco no cronograma, processo, tarefas e estrutura para apoiar o progresso do trabalho.
- *Piecemeal Growth Pattern Language*: descreve como criar a organização e o processo juntos.
- *Organizational Style Pattern Language*: trata do relacionamento entre os papéis na organização.
- *People and Code Pattern Language*: explica o relacionamento entre a estrutura de uma organização e os artefatos que são construídos.

Uma parte dos padrões dessas linguagens foi herdada da linguagem de padrões "*A Generative Development-Process Pattern Language*" (Coplien, 1995). Apesar do título do livro ser "*Organizational Patterns of Agile Software Development*" (Padrões Organizacionais de Desenvolvimento de Software Ágil), o termo "Ágil" foi usado por questões de marketing. Muitos dos padrões dessas linguagens podem contribuir para agilidade, mas o principal objetivo é a efetividade (Coplien et al., 2004).

O padrão *Fire Walls* é apresentado para exemplificar os padrões de Coplien et al. (2004). A forma de apresentação desses padrões é a Alexandrina.

Nome: Fire Walls ** (para Coplien et al. (2004), as estrelas variam entre zero e duas, dependendo da freqüência com que os autores perceberam a aplicação do padrão)

Contexto: uma organização de desenvolvedores está formada em um contexto corporativo ou social, em que os desenvolvedores são examinados por colegas, financiadores, clientes e por outras pessoas externas. Desenvolvedores são freqüentemente distraídos por pessoas externas, que sentem necessidade de passar informações e críticas.

Resumo do Problema: é importante proteger os desenvolvedores de outras pessoas envolvidas no projeto, que não participam do desenvolvimento, mas sentem necessidade de ajudar por meio de comentários ou críticas.

Problema Detalhado: o isolamento não funciona: o fluxo da informação é importante. Mas, o excesso de comunicação aumenta de forma não linear em relação ao número de colaboradores externos.

Solução: crie um cargo de gerente, que protege os desenvolvedores de interações com cargos externos. A responsabilidade desse cargo é "manter as pestes longe".

Contexto Resultante: a nova organização isola os desenvolvedores de interrupções externas insignificantes. Para evitar o isolamento, esse padrão deve ser utilizado em conjunto com outros, como *Engage Customers* e *Gate Keeper*.

Análise Racional: o padrão *Gate Keeper* facilita o fluxo de informações úteis; *Fire Walls* restringe o fluxo de informações. É necessário balancear esses dois padrões.

Beedle et al. (1999) propõem uma linguagem de padrões de extensão para as linguagens de padrões organizacionais já existentes, a "*Scrum Pattern Language*". Nessa linguagem as práticas Scrum, descritas na forma de padrões, são combinadas com alguns padrões organizacionais e de processo de Coplien (1995), para guiar o desenvolvimento de software de forma mais adaptativa e estruturar melhor a organização. Para Beedle (1997), o próprio Scrum é um padrão organizacional.

O padrão *Scrum Meeting* é apresentado a seguir para exemplificar a forma de apresentação dos padrões de Beedle et al. (1999), que descrevem seus padrões com os seguintes elementos: nome, contexto, problema, forças, solução, análise racional, usos conhecidos e contexto resultante.

Nome: *Scrum Meeting*

Contexto: você é um desenvolvedor de software ou gerente de uma equipe de desenvolvimento no qual estão envolvidos: criatividade, descobertas e testes.

Problema: qual é a melhor forma de controlar um processo de desenvolvimento de software, em que é difícil definir os artefatos que serão produzidos e os processos para consegui-los?

Forças: é difícil realizar estimativas exatas para atividades que envolvem descobertas, criatividade ou testes. Planejar e re-priorizar tarefas consome tempo.

Solução: fazer reuniões diárias de aproximadamente quinze minutos, onde se deve discutir o que foi produzido desde a última reunião, que problemas foram encontrados para realizar as tarefas nas últimas vinte e quatro horas e o que será feito nas próximas vinte e quatro horas.

Análise Racional: é muito fácil superestimar ou subestimar esforços, o que leva a um desperdício de tempo ou a um atraso para conclusão de tarefas. É melhor ter um mecanismo adaptativo que fornece uma amostra do que está sendo realizado em pequenos períodos de tempo, ao invés de re-priorizar tarefas constantemente.

Usos Conhecidos: Nike Securities em Chicago e Elementrix Technologies.

Contexto Resultante: melhor visibilidade do status do projeto e da produtividade individual, menos tempo perdido com obstruções e melhor socialização entre os membros da equipe.

3. Visão Geral sobre Métodos Ágeis, Extreme Programming e Scrum

Existe atualmente grande interesse nos métodos modernos de desenvolvimento, conhecidos como métodos ágeis. Esses métodos abordam o processo de desenvolvimento de software de forma diferente dos modelos preconizados anteriormente pela Engenharia de Software, que tinham forte ênfase na documentação e nos processos. A principal diferença está na forma como as mudanças são tratadas durante o desenvolvimento do software. Os modelos de processo convencionais adotam a estratégia de previsibilidade. Eles utilizam técnicas para tentar levantar todos os requisitos e compreender o domínio do problema antes de iniciar o desenvolvimento. Depois de levantados os requisitos, é feito um planejamento para que as mudanças possam ser controladas no decorrer do processo de desenvolvimento do software. Os métodos ágeis optam pela adaptabilidade. Os requisitos são levantados aos poucos e o planejamento é contínuo, para que a adaptação às mudanças possa ocorrer.

Cockburn (2001) define desenvolvimento ágil de software como uma abordagem de desenvolvimento que trata os problemas das mudanças rápidas: mudanças nas forças de mercado, requisitos de sistemas, tecnologia de implementação e equipes de projeto dentro de período de desenvolvimento.

Os métodos ágeis enfatizam os aspectos humanos do desenvolvimento de software ao invés dos aspectos de Engenharia (Lycett et al., 2003). Segundo Highsmith et al. (2001), o que existe de novo nos métodos ágeis não são as práticas que eles usam, mas o reconhecimento de que as pessoas são os principais condutores de sucesso do projeto. Outra característica desses métodos é que eles não são centrados nos artefatos. Eles optam por uma documentação apropriada para evitar redundâncias e excessos, para que auxilie efetivamente o desenvolvimento do software.

As Seções 3.1 e 3.2 apresentam os métodos ágeis XP e Scrum resumidamente.

3.1. Extreme Programming

O XP, criado por Kent Beck e Ward Cunningham, é o mais popular dos métodos ágeis. Ele é indicado para equipes pequenas e médias, com até dez integrantes, que desenvolvem software baseado em requisitos não totalmente definidos e que se modificam rapidamente (Beck, 1999).

As práticas do XP não são novidades: ele reúne práticas de implementação e gerenciamento em um conjunto coerente, acrescentando as idéias de processo.

XP define um conjunto de doze práticas, apresentadas a seguir, escolhidas com base em quatro valores que são: comunicação, simplicidade, *feedback* e coragem.

Jogo do Planejamento (*The Planning Game*). Determina rapidamente o escopo das próximas versões, combinando as prioridades de negócio e as estimativas técnicas.

Pequenas Versões (*Small releases*). A equipe deve colocar rapidamente um sistema simples em produção, uma versão pequena, e depois entregar novas versões em poucos dias ou poucas semanas.

Metáfora (*Metaphor*). Uma metáfora é uma descrição simples de como o sistema funciona. Ela fornece uma visão comum do sistema e guia o seu desenvolvimento.

Projeto simples (*Simple design*). O sistema deve ser projetado o mais simples possível. Complexidade extra é removida assim que descoberta.

Testes (*Testing*). Os programadores escrevem testes de unidade continuamente. Esses testes são criados antes do código e devem ser executados perfeitamente para que o desenvolvimento continue. Os clientes também escrevem testes para validar se as funções estão finalizadas.

Refatoração (*Refactoring*). Os programadores reestruturam o sistema durante todo o desenvolvimento, sem modificar seu comportamento externo (Fowler, 1999). Isso é feito para simplificar o sistema, adicionar flexibilidade ou melhorar o código.

Programação pareada (*Pair programming*). Todo código produzido é feito em pares, duas pessoas trabalhando em conjunto na mesma máquina.

Propriedade coletiva (*Collective ownership*). Qualquer um pode alterar qualquer código em qualquer momento, o código é de propriedade coletiva.

Integração contínua (*Continuous integration*). Uma nova parte do código deve ser integrada assim que estiver pronta. Consequentemente, o sistema é integrado e construído várias vezes ao dia.

Semana de 40 horas (*40-hour week*). XP defende um ritmo de trabalho que possa ser mantido, sem prejudicar o bem estar da equipe. Trabalho além do horário normal pode ser necessário, mas fazer horas extras por períodos maiores que uma semana é sinal de que algo está errado com o projeto.

Cliente junto aos desenvolvedores (*On-site customer*). Os desenvolvedores devem ter o cliente disponível todo o tempo, para que ele possa responder às dúvidas que os desenvolvedores possam ter.

Padronização do Código (*Coding standards*). Os programadores escrevem o código seguindo regras comuns enfatizando a comunicação por meio do código.

Alguns dos papéis identificados em XP: Programador (*Programmer*), Cliente (*Customer*), Testador (*Tester*), Investigador (*Tracker*), Orientador (*Coach*), Consultor (*Consultant*) e Gerente (*Manager*). Ressalta-se que o testador não é necessariamente uma pessoa que realiza somente essa atividade (Beck, 1999).

As práticas do XP apóiam umas às outras, devem ser usadas em conjunto e todas devem ser aplicadas para se ter agilidade no processo. Aplicar as práticas de forma isolada pode não produzir a agilidade desejada.

3.2. Scrum

O Scrum foi desenvolvido para gerenciar o processo de desenvolvimento de software em ambientes em que os requisitos estão em constante mudança. Ele é apropriado para equipes pequenas, com até dez integrantes. (Abrahamsson et al., 2002). Schwaber et al. (2002) sugerem que a equipe contenha de cinco a nove integrantes.

O Scrum não exige ou fornece métodos ou práticas específicas de desenvolvimento de software, mas exige certas práticas de gerenciamento, que são descritas por Abrahamsson et al. (2002):

Tarefas do Produto (*Product Backlog*): define tudo o que é necessário no produto final. Contém uma lista priorizada e constantemente atualizada dos requisitos do sistema que está sendo construído ou otimizado.

Estimativa de esforço (*Effort Estimation*): como o Scrum é um processo iterativo a estimativa de esforço para realizar as tarefas deve ser realizada frequentemente.

***Sprint*:** procedimento de adaptação às mudanças de variáveis de ambiente, como requisitos, tempo, recursos ou tecnologia. S*prints* são intervalos fixos de tempo, em que todo o trabalho é realizado. No Scrum um *sprint* tem duração de trinta dias. Durante um *sprint* a equipe Scrum se organiza para produzir um incremento do produto. Essa prática contém: reuniões de planejamento dos *sprints* (*Sprint Planning Meetings*), para decidir os objetivos e funcionalidades do próximo *sprint*; Tarefas do *Sprint* (*Sprint Backlog*), que é uma lista de itens de trabalho de produto selecionados para o próximo *sprint*; Reuniões Scrum diárias (*Daily Scrum Meetings*), de aproximadamente quinze minutos realizadas para verificar o progresso do projeto e para discutir questões como: o que foi feito desde a última reunião e o que precisa ser feito até a próxima.

Reunião de Revisão de *Sprint* (*Sprint Review Meeting*): no último dia do *sprint*, os resultados são apresentados.

Segundo Abrahamsson (2002) os papéis identificados no Scrum são: Mestre (*Scrum Master*), Proprietário do produto (*Product Owner*), Equipe Scrum (*Scrum Team*), Cliente (*Customer*) e Gerência (*Management*).

Por não fornecer métodos e práticas específicas de desenvolvimento, o Scrum se torna um método mais flexível, já que o desenvolvimento pode ser tratado da forma que for melhor para a organização.

4. Métodos Ágeis e Padrões Organizacionais e de Processo

No estudo realizado foi possível detectar que padrões organizacionais e de processo e métodos ágeis estão diretamente relacionados. Sutherland (2003) afirma que alguns padrões de Coplien (1995) influenciaram o desenvolvimento de Scrum, enquanto Beck (1999) declara que muitas das idéias do XP provêm da linguagem de padrões Episodes (Cunningham, 1996).

Assim, muitas idéias encontradas nos padrões organizacionais e de processo também são encontradas nos métodos ágeis. Por exemplo, no XP o cliente define a funcionalidade do sistema a ser desenvolvido por meio das chamadas estórias do usuário, e as prioriza em seguida. Na linguagem de padrões Episodes, os padrões *Implied Requirement* e *Work Queue* sugerem, respectivamente, que a funcionalidade seja identificada e depois priorizada. Além desses, outros padrões organizacionais e de processo podem ser destacados pela relação que possuem com os métodos ágeis.

Com base nos estudos realizados, a Tabela 1 mostra alguns padrões organizacionais e de processo que são usados como prática nos métodos ágeis, porém não existe referência para eles nos métodos. A seleção foi realizada com base nos conceitos comuns encontrados nos métodos ágeis e nesses padrões. A coluna 1 mostra o nome do padrão; na coluna 2 consta a(s) linguagem(ns) de padrões a(s) qual(is) o padrão pertence; um breve resumo do problema que o padrão resolve é apresentado na

coluna 3 e a coluna 4 indica o(s) método(s) ágil (eis) que usa(m) o padrão como prática.

Tabela 1. Padrões Organizacionais e de Processo e Métodos Ágeis

Nome	Linguagem de Padrões	Resumo	Método Ágil
Implied Requirement	*Project Management Pattern Language, Episodes.*	O problema é definir as necessidades do cliente de forma significativa para os desenvolvedores. Portanto, selecione e nomeie partes de funcionalidade e crie uma lista com essas partes.	XP, Scrum
Work Queue	*Project Management Pattern Language, Episodes.*	O problema é conceder tempo para realizar tudo. Portanto, crie um cronograma que é simplesmente uma lista priorizada de trabalho. Use a lista do *Implied Requirement* como ponto de partida e ordene-a, em uma ordem de implementação, de modo que favoreça os itens mais urgentes ou de maior prioridade.	XP, Scrum
Size the Organization	*Piecemeal Growth Pattern Language*	Quando a equipe de desenvolvimento é muito grande, raramente os projetos são entregues dentro do prazo e orçamento previstos. Se a equipe é muito grande a comunicação pode falhar. Se a equipe é pequena, a produtividade vai diminuir. Por isso, escolha aproximadamente dez pessoas para compor a equipe de desenvolvimento e evite acrescentar indivíduos depois de iniciado o desenvolvimento.	XP, Scrum
Engage Customers	*Piecemeal Growth Pattern Language*	Se você quer gerenciar um processo de desenvolvimento incremental que acomode informações fornecidas pelo cliente, junte o cliente aos desenvolvedores e arquitetos, não somente ao QA (*Quality Assurance*) ou marketing.	XP, Scrum
Developing in Pairs	*Piecemeal Growth Pattern Language*	Se você quer melhorar a efetividade individual dos desenvolvedores, coloque os desenvolvedores para trabalharem em pares.	XP
Few Roles	*Organizational Style Pattern Language*	As pessoas em um projeto devem se comunicar para o projeto progredir. Mas, o custo indireto dessa comunicação pode impedir o verdadeiro progresso que ela deveria facilitar. Portanto, tente manter o número de papéis da organização abaixo de dezesseis.	XP, Scrum
Stand up Meeting	*People and Code Pattern Language*	Em tempos de mudanças rápidas é essencial que todos os membros da organização recebam as mesmas informações. Portanto, realize reuniões diárias, de aproximadamente	XP, Scrum

		quinze minutos, com a equipe, para trocar informações sobre o projeto.	
Developer Controls Process	*Project Management Pattern Language*	Como os Desenvolvedores contribuem diretamente no desenvolvimento dos artefatos visíveis para o usuário final, faça do desenvolvedor o ponto foco de informação do processo.	XP, Scrum
Patron Role	*Piecemeal Growth Pattern Language*	É importante dar continuidade ao projeto. Mas, um controle centralizado pode dificultar o progresso. Assim, eleja um "Patrono" para o projeto, para que as barreiras que impedem o progresso do projeto sejam removidas.	Scrum
Surrogate Customer	*Piecemeal Growth Pattern Language*	É importante trocar idéias com os clientes. Mas se o cliente não estiver disponível, crie um papel de Substituto do Cliente no projeto, com alguém que irá tentar pensar como o cliente.	XP
Fire Walls	*Piecemeal Growth Pattern Language*	É importante proteger os desenvolvedores de outras pessoas envolvidas no projeto, que não participam do desenvolvimento, mas sentem necessidade de ajudar por meio de comentários ou críticas. Portanto, crie um cargo de gerente, que protege os desenvolvedores de interações com cargos externos.	Scrum

Tanto no XP quanto no Scrum as funções que irão compor a versão final do software são selecionadas e depois priorizadas. Essas práticas são as propostas pelos padrões *Implied Requirement* e *Work Queue*. Da mesma forma que o padrão *Size the Organization* propõe, os dois métodos defendem o uso de equipes pequenas ou médias. Um dos valores que serve como base para os métodos ágeis é a colaboração dos clientes. O padrão *Engage Customers* sugere que os clientes devem estar próximos dos desenvolvedores e arquitetos e não só da garantia de qualidade e marketing. Com os clientes participando, o desenvolvimento se torna mais rápido, pois para conseguir agilidade os desenvolvedores precisam de respostas rápidas dos clientes. Uma das práticas do XP é a programação em pares, em que todo código produzido é feito em pares, trabalhando juntos na mesma máquina. O padrão *Developing in Pairs* sugere que em pares os desenvolvedores produzem mais do que a soma dos dois individualmente. Cockburn et al. (2000) afirmam que a programação em pares traz importantes benefícios ao desenvolvimento de software. Os programadores trabalham mais rápido, aprendem mais sobre o projeto e o sistema, o código desenvolvido é menor e o software é produzido com menos defeitos. Poucos papéis são definidos no XP e no Scrum.

Ambos seguem o padrão *Few Roles*, que sugere que o número de papéis em uma organização seja de aproximadamente de dezesseis ou menos. *Stand up Meetings* são reuniões rápidas de aproximadamente quinze minutos, que é uma das práticas de gerenciamento do XP e do Scrum, realizada para avaliar o progresso alcançado e planejar as próximas atividades. Nos dois métodos, o desenvolvedor é o ponto central do projeto, o que é sugerido pelo padrão *Developer Controls Process.* O Padrão *Patron*

Role e o Padrão *Fire walls* propõem, respectivamente, que sejam criados papéis para dar continuidade ao projeto e proteger o desenvolvedores de interações externas. No Scrum isso é responsabilidade do *Scrum Master*. Outra semelhança pode ser encontrada no padrão *Surrogate Customer* e na prática "Cliente junto aos desenvolvedores" (*On-site customer*) do XP. O XP defende que o cliente deve estar disponível todo o tempo, para que os desenvolvedores possam tirar dúvidas sobre o projeto. Entretanto, se não for possível ter o cliente disponível, alguém da equipe deve desempenhar esse papel. Essa prática é proposta pelo padrão *Surrogate Customer*.

Além dos padrões apresentados na Tabela 1, outros podem ser integrados aos métodos ágeis para melhorar ou adaptar o método ágil, conforme mostrado na Tabela 2. Esses padrões foram identificados com base nos princípios dos métodos ágeis, e propõem soluções que facilitam a comunicação, interação e adaptação às mudanças que podem ocorrer no projeto.

Tabela 2. Padrões Organizacionais e de Processo

Nome	Linguagem de Padrões	Resumo
Community of Trust	*Project Management Pattern Language*	O relacionamento social tem um impacto significante na efetividade da equipe. Faça suas atividades de forma que demonstre confiança explicitamente. As ações devem ser visíveis e evidentes, para que as pessoas da equipe confiem umas nas outras.
Self Selecting Team	*Piecemeal Growth Pattern Language*	Não existe um critério perfeito para selecionar membros de uma equipe, mas os interesses dos indivíduos não devem ser ignorados. Assim, crie equipes entusiasmadas com as pessoas escolhendo sua própria equipe.
Unity of Purpose	*Piecemeal Growth Pattern Language*	Muitos projetos têm um inicio difícil com as pessoas se esforçando para trabalharem juntas. Frequentemente, as pessoas têm idéias diferentes de como o produto final deveria ser. Assim, o líder do projeto deve expor para todos membros da equipe uma visão comum e propósito geral.
Matron Role	*Piecemeal Growth Pattern Language*	Algumas atividades são necessárias para manter a equipe prosseguindo no trabalho técnico. Por isso, assegure que a equipe contenha uma "Mãe", que vai tratar dos assuntos sociais e pessoais necessários para manter a equipe unida.
Compensate Success	*Piecemeal Growth Pattern Language*	Estabeleça recompensas para os indivíduos que contribuem para o sucesso do projeto. Toda a equipe deve receber recompensas parecidas, para evitar desmotivação individual. Assim, a organização fica mais focada na satisfação do cliente e no sucesso do sistema.
Organization Follows Location	*Organizational Style Pattern Language*	Se for necessário distribuir o trabalho geograficamente, a comunicação pode ser prejudicada, mas você pode limitar os danos se o trabalho puder ser dividido.

		Assim, a divisão de tarefas deve estar de acordo com a distribuição geográfica dos envolvidos no projeto. Responsabilidades devem ser atribuídas de forma que decisões possam ser tomadas localmente.
Face To Face Before Working Remotely	*Organizational Style Pattern Language*	A distância geográfica dificulta a comunicação. Assim, inicie um projeto distribuído com uma reunião cara a cara com todos para que se estabeleça uma uniformidade no projeto.
Standards Linking Locations	*Organizational Style Pattern Language*	O isolamento de desenvolvedores não deve ocorrer em projetos geograficamente distribuídos. A equipe deve se comunicar por meio de interfaces definidas, o código deve interagir. Assim, utilize normas para representar tudo o que está relacionado à arquitetura.
Shaping Circulation Realms	*Organizational Style Pattern Language*	A comunicação entre os participantes do projeto é fundamental para o sucesso e não se pode esperar que a comunicação aconteça espontaneamente. Portanto, crie estruturas na organização ou no espaço de trabalho que apóiem a comunicação.
The Water Cooler	*Organizational Style Pattern Language*	As organizações precisam evitar o isolamento das equipes. Em ambientes amplos é difícil apoiar a freqüente interação entre as equipes. Promova estruturas sociais que não estão relacionadas ao local de trabalho, onde as pessoas podem se encontrar, tanto para pausa quanto para comunicação.

Os padrões *Community of Trust, Self Selecting Team, Unity of Purpose* e *Compensate Success* abordam questões relacionadas aos indivíduos envolvidos no processo de desenvolvimento. Esses padrões podem ser integrados aos métodos ágeis para aumentar a motivação individual, melhorar o relacionamento e bem estar dos envolvidos no desenvolvimento e, conseqüentemente, agilizar o desenvolvimento de software.

Em uma pesquisa realizada para ensinar XP para estudantes, Goldman et al. (2004) destacam alguns aspectos importantes que devem ser observados quando se adota o XP como processo de desenvolvimento. Foi observado que fornecer lanches simples para os estudantes é uma forma eficiente de mantê-los no desenvolvimento do software por um longo período. Assim, os estudantes permaneciam concentrados no desenvolvimento. Esse é o caso de aplicação do padrão *Matron Role*. Outro ponto destacado foi que os desenvolvedores foram organizados seguindo diretrizes para facilitar a comunicação. Os padrões *Shaping Circulation Realms* e *The Water Cooler* são aplicados com esse objetivo nas organizações.

Em um estudo realizado sobre programação pareada, Baheti et al. (2002) mostram que é possível desenvolver software com programação pareada distribuída. Porém, com os programadores geograficamente divididos, a falta de comunicação pode afetar o projeto. Para tratar o problema da comunicação em ambientes distribuídos, os padrões *Organization Follows Location, Standards Linking Locations* e *Face To Face*

Before Working Remotely podem ser aplicados para que a agilidade não seja afetada por falta de comunicação.

Assim, aspectos importantes do desenvolvimento ágil de software são abordados pelos padrões apresentados. A integração de métodos ágeis e de padrões organizacionais e de processo se torna cada vez mais necessária, possibilitando um desenvolvimento de software mais rápido e com qualidade.

5. Considerações Finais

Os padrões organizacionais e de processo apóiam de forma efetiva a construção do software. Assim, se integrados com métodos ágeis, o software pode ser desenvolvido de forma mais rápida e com mais qualidade, pois os padrões são soluções de sucesso para problemas recorrentes que podem ser utilizados tanto para melhorar quanto para adaptar os métodos ágeis.

Uma questão que surge quando os padrões organizacionais e de processo são abordados é como integrá-los aos métodos de desenvolvimento de software.

A linguagem de padrões apresentada por Beedle et al. (1999) é um exemplo de integração de alguns padrões organizacionais e de processo com o método ágil Scrum. Por meio dessa combinação, Beedle et al. (1999) descrevem de forma mais clara a estrutura da organização de desenvolvimento de software. Dentre os padrões integrados com o Scrum, pode-se destacar dois que foram apresentados na Tabela 1: o *Fire Walls* e o *Developer Controls Process*. O padrão *Fire Walls* está relacionado ao *Scrum Master*, que deve filtrar as informações irrelevantes que não contribuem para melhoria do projeto. Outro relacionamento é o do padrão *Developer Controls Process* com o *Scrum Team*, que é a equipe de desenvolvimento. Os desenvolvedores, que tem autoridade para decidir ações necessárias para alcançar seus objetivos, são os pontos chave de comunicação no projeto, por estarem em melhor posição para assumir responsabilidade pelo produto.

Apesar deste estudo sobre padrões mostrar a existência de alguns padrões organizacionais e de processo que podem melhorar o desenvolvimento de software, existe ainda o desafio de integrar esses padrões a um método ágil (Scrum ou XP) e estabelecer um processo de desenvolvimento de software ágil, com soluções de sucesso em nível organizacional e de processo.

6. Referências Bibliográficas

Abrahamsson, P.; Salo, O.; Ronkainen, J.; Warsta, J. *Agile Software Development Methods: Reviews and Analysis*. Espoo: VTT Publications, 2002. Disponível em: <http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf>. Acesso em: 11 mar. 2005.

Baheti, P.; Williams, L.; Gehringer, E.; Stotts, D. *Exploring Pair Programming in Distributed Object-Oriented Team Projects*. In Proceedings of XP/Agile Universe 2002, Chicago, Agosto, 2002.

Beck, K. *Extreme Programming Explained – Embrace Change*. Addison-Wesley. 1999.

Beck, K.; Beedle, M.; Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R.; Mellor, S.; Schwaber, K.; Sutherland, J.; Thomas, D. *Manifesto for Agile Software*

Development. 2001. Disponível em: <http://www.agilemanifesto.org/>. Acesso em: 20 fev. 2005.

Beedle, M. *Scrum is an Organization Pattern*. 1997. Disponível em: <http://www.jeffsutherland.org/scrum/scrum_pattern.html>. Acesso em: 23 mar. 2005.

Beedle, M.; Devos, M.; Sharon, Y.; Schwaber, K.; Sutherland, J. SCRUM: *An Extension Pattern Language for Hyperproductive Software Development.* In: Harrison, N.; Foote, B.; Rohnert, H. Pattern Languages of Program Design 4. Addison-Wesley, 1999.

Cockburn, A.; Williams, L. *The Costs and Benefits of Pair Programming.* In Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000), Junho, 2000.

Cockburn, A.; Highsmith, J. *Agile Software Development: The People Factor*. IEEE Computer, 2001.

Cockburn, A. *Agile software development*. Boston: Addison Wesley, 2002.

Coplien, J. O. *A Generative Development-Process Pattern Language*. In: Coplien, J.; Schmidt, D. *Pattern Languages of Program Design*. USA: Addison-Wesley, 1995.

Coplien, J. O.; Schmidt, D. C. *Pattern Languages of Program Design*. Reading – MA, USA: Addison-Wesley, 1995.

Coplien, J. O.; Harrison N. B. *Organizational Patterns of Agile Software Development*. 1. ed. Prentice Hall, 2004.

Cunningham, W. *Episodes: A Pattern Language of Competitive Development*. In: Vlissides, J.; Coplien, J.; Kerth, N. Pattern Languages of Program Design 2, Addison-Wesley, 1996.

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Fowler, M. *The New Methodology*. 2003. Disponível em: <http://www.martinfowler.com/articles/newMethodology.html>. Acesso em: 15 jun 2005.

Goldman, A.; Kon, F; Silva, P. J. S.; Yoder J. W. *Being Extreme in the Classroom: Experiences in Teaching XP.* In Journal of the Brazilian Computer Society, volume 10, número 2, pp. 1-17. Novembro, 2004.

Highsmith, J.; Cockburn, A. *Agile Software Development: The Business of Innovation*. IEEE Computer, 2001.

Kerth, H.; Cunningham, W. *Using Patterns to Improve our Architectural Vision*. IEEE Software, v.14, n. 1, p. 53-59, 1997.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3. ed. Prentice Hall, 2004.

Lycett, M.; Macredie, R. D.; Patel, C.; Paul, R. J. *Migrating Agile Methods to Standardized Development Practice*. In: Computer, pp. 79-85. IEEE Computer Society. Jun. 2003.

Schwaber, K.; Beedle, M. *Agile Software Development with SCRUM*. Prentice-Hall, 2002.

Sutherland, J. SCRUM: *Another Way to Think about Scaling a Project*. 2003. Disponível em: <http://jeffsutherland.org/scrum/2003_03_01_archive.html>. Acesso em: 20 mar. 2005.

Taber, C.; Fowler, M. *An iteration in the life of an XP project*. Cutter IT Journal, 13(11), Novembro, 2000. Versão eletrônica disponível em: <http://www.martinfowler.com/articles/planningXpIteration.html>. Acesso em: 20 set. 2005.

Cooperação entre Padrões de Projeto na Resolução de Problemas de Processamento de Imagens Baseados em Filtros de Convolução

Daniel Welfer[1], **Marcos Cordeiro d'Ornellas**[1]

[1]Departamento de Eletrônica e Computação
Universidade Federal de Santa Maria(UFSM)
97.105-900, Campus Universitário - Centro de Tecnologia
Santa Maria, RS - Brasil

welfer@gmail.com, ornellas@inf.ufsm.br

Abstract. *This paper presents a solution to implement convolution filters using design patterns. Convolution filters are used to solve several problems in the field of image processing, such as edges enhancement, mathematical morphology and smoothing of noisy images. However, the implementation of convolution filters is very complex because a filter can vary both in type and size. In order to manage this complexity, a cooperation of design patterns was used in the construction of a software component with the purpose of codifying these types of filters. This approach has assured some important characteristics, such as legibility, easy maintenance and reusability of source-code.*

Resumo. *Este trabalho apresenta uma solução para implementar filtros de convolução através da aplicação de padrões de projeto. Filtros de convolução são utilizados para resolver vários problemas na área do processamento de imagens, tais como realce de bordas, morfologia matemática e suavização de imagens ruidosas. Entretanto, sua implementação é muito complexa uma vez que um filtro pode variar tanto em tipo como em tamanho. Dessa forma, para gerenciar essa complexidade, foi utilizado uma cooperação entre padrões de projeto na construção de um componente de software para codificar esses tipos de filtros. Essa abordagem assegurou características como legibilidade, fácil manutenção e reusabilidade de código-fonte.*

1. Introdução

Atualmente, o processamento e a análise de imagens estão sendo empregados nas mais diferentes áreas do conhecimento. Na área médica, por exemplo, as imagens são utilizadas para diagnosticar patologias. No domínio geoespacial, elas são utilizadas para visualizar o estado climático de uma região ou até mesmo para registrar o relevo de outros planetas [Gosling, 2004], [Akre and Tabirca, 2003]. No campo comercial, as imagens estão cada vez mais presentes no cotidiano das pessoas através das câmeras digitais e scanners cada vez mais portáteis.

Porém, as imagens digitais, normalmente são dependentes de um software que gerencia todo o seu processamento ou análise. E é, justamente nesse contexto, que os padrões de projeto são utilizados. Os padrões de projeto são formas de construção de software comprovadamente funcionais que garantem legibilidade, fácil manutenção e reutilização de código-fonte no desenvolvimento de alguma aplicação computacional.

Nesse trabalho, esses padrões são utilizados para implementar uma solução adaptável para o problema dos filtros de convolução. Para isso, na seção 2, é apresentado uma noção sobre esses filtros através de um processo conhecido como segmentação de imagens. Na seção 3, é descrito a colaboração entre padrões de projeto na concepção de um componente de software para a solução do problema dos filtros de convolução. Após é descrito as conclusões finais sobre o trabalho.

2. Filtros de Convolução: noções e aplicação

Para esclarecer o emprego dos filtros de convolução, que aparecem com muita freqüência em processamento e análise de imagens digitais, será utilizado o problema do realce de bordas no processo de segmentação de imagens.

Segmentação de imagens é o processo pelo qual se particiona uma imagem em um conjunto de regiões similares colocando-as em primeiro ou segundo plano conforme o objeto de interesse [Ritter and Wilson, 2000], [Russ, 1998]. Esse processo, típico de sistemas de visão computacional, baseia-se na propriedades dos pixels que formam a imagem, isto é, a intensidade que cada um apresenta em sua respectiva banda. Dessa forma, em uma imagem com valores de pixels no intervalo entre 0 e 255 a borda é detectada quando há mudanças bruscas nos níveis de cinza, ou seja, em sua magnitude. Segundo Gonzalez e Woods, [Gonzalez and Woods, 1992], essa é uma abordagem baseada na descontinuidade que, por sua vez, precisa ser complementada por um processo também de segmentação conhecido como binarização. Esse último passo, também conhecido como limiarização ou thresholding, faz-se necessário para identificar quais são os pixels da borda dos demais.

Nesse trabalho, a detecção de bordas foi aplicado às imagens de banda simples, necessitando assim, que o aplicativo desenvolvido execute a conversão da imagem captada, que por sua vez possui um modelo de cores baseado em três bandas, isto é, RGB para o modelo monocromático.

Uma grande variedade de técnicas são usadas para computar as bordas de uma imagem [Ritter and Wilson, 2000]. Dentre elas, foi utilizada a técnica da transformada que aproxima o gradiente que, segundo Gonzalez e Woods, [Gonzalez and Woods, 1992] é o método mais comum de diferenciação em aplicações de processamento de imagens. Para isso, são utilizados filtros, conhecidos como máscaras ou *kernels* de convolução. Eles recebem essa denominação, porque operam unicamente no domínio do espaço. Assim, para a detecção dessas bordas, são necessárias duas convoluções na imagem original. A primeira operação de convolução detecta as bordas na direção horizontal e a segunda na direção vertical, formando assim, as bases do subespaço de bordas. A literatura apresenta vários desses filtros como o de Roberts, Prewit, Sobel e Frei e Chen. Na figura 1, é demonstrado as máscaras de Frei e Chen na forma de arranjo bidimensional.

Pode-se observar na figura 1, que os coeficientes da máscara somam zero, o que significa dizer que em áreas constantes a resposta será nula(ganho zero) ocorrendo assim, a diferenciação entre as regiões, isto é, dos pixels que pertencem a borda dos demais [Ritter and Wilson, 2000]. Após a aplicação dessas máscaras, utiliza-se um limiar ou *thresholding* global sob a imagem segmentada. Esse é um passo bastante conveniente uma vez que o algoritmo de Frei e Chen não consegue definir valores exclusivos para as bordas, isto é, ainda não é conhecido quais são realmente os pixels das bordas da imagem [Ritter and Wilson, 2000]. Dessa forma, com a limiarização obtém-se uma imagem binária cujo plano de fundo apresentava valor 0 e as bordas valor 1. Assim, para a imagem f(x,y), seu limiar T(x,y) foi encontrado como mostra a equação da figura 2:

–1.0	–1.414	–1.0
0.0	0.0	0.0
1.0	1.414	1.0

a)

1.0	0.0	–1.0
1.414	0.0	–1.414
1.0	0.0	–1.0

b)

Figura 1: O *kernel* de convolução de Frei e Chen : a) filtro vertical e b) filtro horizontal.

$$\text{T(x,y)} = \begin{cases} 0 & se \quad f(x,y) < k \\ 1 & se \quad f(x,y) \geq k \end{cases}$$

Figura 2: Modelo para encontrar o Thresholding, onde k é o limiar especificado pelo usuário.

Esse processo de segmentação foi utilizado em imagens de componentes industriais. A idéia era detectar os contornos dessas peças e gerar um arquivo texto contendo as coordenadas espaciais dessas bordas. Através desse arquivo a peça pode ser reconstruída em ambiente assistido por computador. A figura 3 demonstra a parte de processamento de imagens desse trabalho. Sublinha-se que, a imagem original depois de ser convertida para tons de cinza sofreu o processo de equalização para fins de melhorar a distribuição da sua luminosidade e, conseqüentemente tornar mais nítida suas bordas.

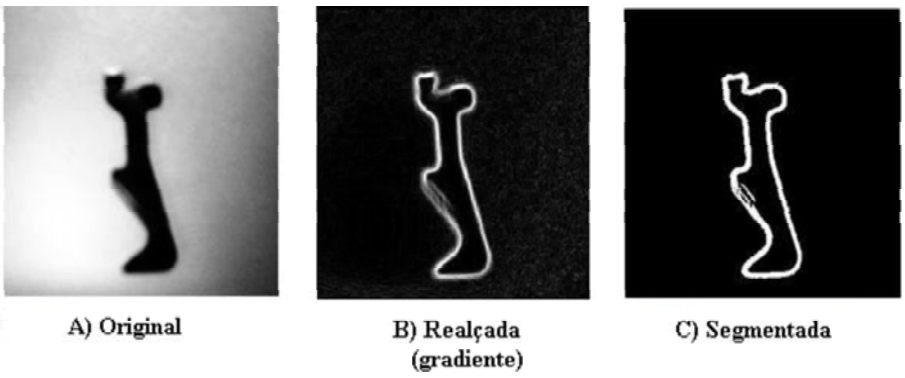


Figura 3: As várias etapas necessárias para detectar a borda.

3. Os padrões utilizados para implementar as máscaras de convolução

De conhecimento da técnica no que diz respeito ao processamento de imagens necessário para esse tipo de segmentação, o próximo passo é projetar o componente de software que automatiza esse processo. Porém, para assegurar as características anteriormente citadas como o reuso e legibilidade foi aplicado uma abordagem de codificação baseada em padrões. Esses padrões são apresentados pela literatura, principalmente, pelos catálogos de Gamma *et al.* [E. Gamma and Vlissides, 1995] e Buschmann *et al.* [Buschmann et al., 1996]. Nesse artigo são utilizados quatro padrões de construção Gamma: o *Builder*, *Strategy*, *Template Method* e *Command*.

3.1. O padrão de criação *Builder* no desenvolvimento de software para processamento de imagens

O padrão *Builder* tem a finalidade de separar um objeto de todas as partes responsáveis por sua formação. Ele é usado quando o objeto possui um grau de complexidade muito grande e para isso sua construção ocorre em etapas, daí o nome bastante intuitivo desse padrão, isto é, construção. O objeto complexo vai sendo construído por partes, sendo que cada uma dessas partes representam um algoritmo distinto que, por sua vez, contrói o objeto final também chamado pela literatura de produto. A vantagem direta desse padrão é a modularidade que apresenta, pois cada uma dessas separações entre etapas e o produto final pode ser implementado através de classes distintas. Outra vantagem citada por Erich Gamma *et al.* [E. Gamma and Vlissides, 1995], diz respeito ao controle que o processo de criação possui, o que já não acontece com o padrão *Abstract Factory* por exemplo. Isso é bem intuitivo de ser entendido pois os demais padrões de projeto de criação, originam os produtos de uma só vez, enquanto que o *Builder* implementa uma lógica mais precisa e incremental.

O padrão *Builder*, assim como o *Factory Method*, que é outro padrão de criação especificado por Gamma *et al.*, também retorna um objeto a partir de uma dada requisição. Porém, como descrito anteriormente, ele possui a vantagem de construir esse objeto a ser retornado de forma mais modular, isto é, passo-a-passo. Outra vantagem é que, o objeto complexo que está sendo construído fica separado de sua representação [Metsker, 2002].

No entanto, antes de usar esse padrão de projeto é necessário enfatizar três aspectos essenciais no que diz respeito ao funcionamento do componente de segmentação baseado em realce de bordas que está sendo explicado:

1. Há diferentes filtros de realce. Assim, o projeto do componente deve permitir que o usuário possa escolher dinamicamente qual dessas máscaras será utilizada;
2. Só escolher qual o filtro utilizar não basta. O Componente também deve oferecer diferentes tamanhos para essas máscaras. Normalmente, na literatura, elas são 3x3 ou 8 conectado (como a figura 1), mas também a literatura descreve como possibilidade de uso máscaras de tamanho 5x5 e 7x7. Dessa forma, é possível ter um mesmo algoritmo operando sob diferentes tamanhos de máscaras.
3. Após escolhida a máscara faz-se necessário ainda executar a operação de convolução propriamente dita. Para isso já se deve ter decidido o nome do filtro e seu tamanho. Nessa etapa de convolução são utilizadas as máscaras horizontais e verticais escolhidas anteriormente.

Nesse contexto, a principal vantagem do padrão de criação *Builder* na implementação desse componente é, justamente gerenciar essas várias partes necessárias para compor um objeto. Sua configuração estrutural é mostrada na figura 4.

O padrão *Builder* representado pela figura 4 possui um nível a mais que o *Factory Method*. A classe `EdgeDetection` é uma classe abstrata que tem como função abrigar diferentes implementações de filtros segundo um mesmo escopo de operação. Suas classes bases são `RobertsSegmentation`(que implementa o filtro de Roberts), `SobelSegmentation`(que implementa o filtro de Sobel), `PrewittSegmentation` (referente ao filtro de Prewitt) e `FreiAndChenSegmentation`(referente ao filtro de Frei e Chen). Cada uma dessas classes possui o método `makeMask()` que tem como parâmetro de entrada o tamanho da máscara que o usuário optou em utilizar e o método `makeSegmentation()` que tem como parâmetro de entrada o valor do limiar escolhido e como valor de retorno a imagem já segmentada. Ambas operações são abstratas e surgem nas sub-classes porém com valores diferentes no que se refere a sua máscara. A primeira operação constrói a máscara tomando como base seu tamanho e a segunda

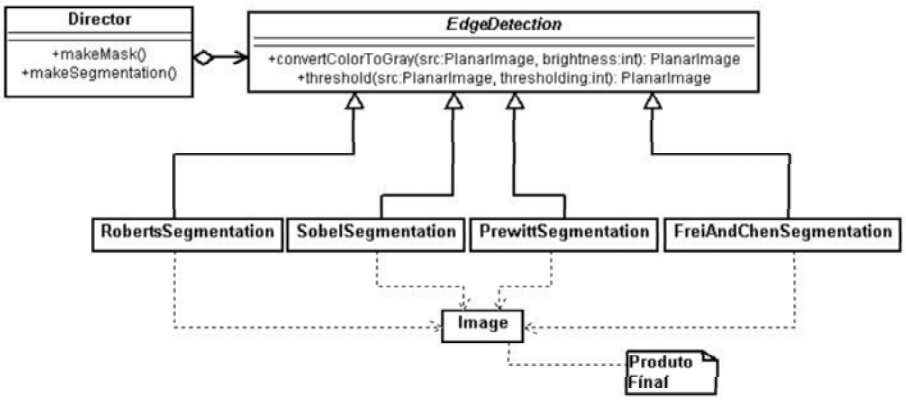


Figura 4: O modelo estrutural do padrão *Builder*. O objeto `Image` é o produto que está sendo construído.

realiza a operação de convolução e depois de binarização. A classe `Image` representa o produto complexo que se está querendo produzir (que no caso é a imagem segmentada). A classe `Director` faz parte da especificação do padrão *Builder* e é a responsável por invocar todos os métodos presentes nas classes filhas em um único método chamado de `Constructor`.

A classe `Image` recebe a imagem a ser segmentada. É essa imagem original que se transformará no produto final. Por isso que ela possui uma relação de dependência entre as classes filhas. As classes filhas dependem do produto final pois, este, em seu estado inicial, carrega a imagem a ser processada pela lógica algorítmica contida no interior dessas classes concretas.

3.2. O padrão comportamental *Strategy* no desenvolvimento de software para processamento de imagens

O *Strategy* é um padrão que encapsula os detalhes de escolha de determinado algoritmo em um contexto onde podem existir famílias inteiras de algoritmos. Dessa forma, para gerenciar qual algoritmo de realce de bordas deve ser invocado, foi utilizado o padrão de projeto comportamental *Strategy*. A intenção desse padrão pode ser deduzida do seu próprio nome, isto é, escolher qual estratégia (que é o algoritmo) deve-se utilizar. No entanto, essa estratégia deve ser exclusiva, isto é, ou invoca o algoritmo de *Frei e Chen* ou de *Roberts* e assim consecutivamente. Para tal finalidade, o padrão *Strategy* encapsula em seu interior a definição de uma família de algoritmos referentes a essas diferentes máscaras de convolução, de forma que sejam intercambiáveis [E. Gamma and Vlissides, 1995].

Segundo James Cooper, [Cooper, 1998], esse padrão também é muito utilizado para comprimir arquivos utilizando diferentes algoritmos, capturar video utilizando diferentes esquemas de compressão e outros. Nesse contexto, esse padrão executa o papel da fábrica onde ocorre a manufatura de determinado objeto e que, por sua vez, apresentam responsabilidades distintas. Essa é a conexão que ele apresenta com o padrão *Builder*, isto é, é através do objeto implementado pelo padrão *Strategy* que será invocado determinada classe filha do padrão *Builder* e que, por sua vez, armazena toda a lógica de determinado filtro de convolução.

A figura 5 mostra o projeto completo do componente isto é, os objetos do padrão *Builder* com a classe que gerencia a chamada dos algoritmos denominada de `Strategy`.

3.3. O padrão comportamental *Template Method*

O outro padrão utilizado no desenvolvimento desse componente foi o *Template Method*. O padrão Template Method possui a finalidade de prover uma lógica comum

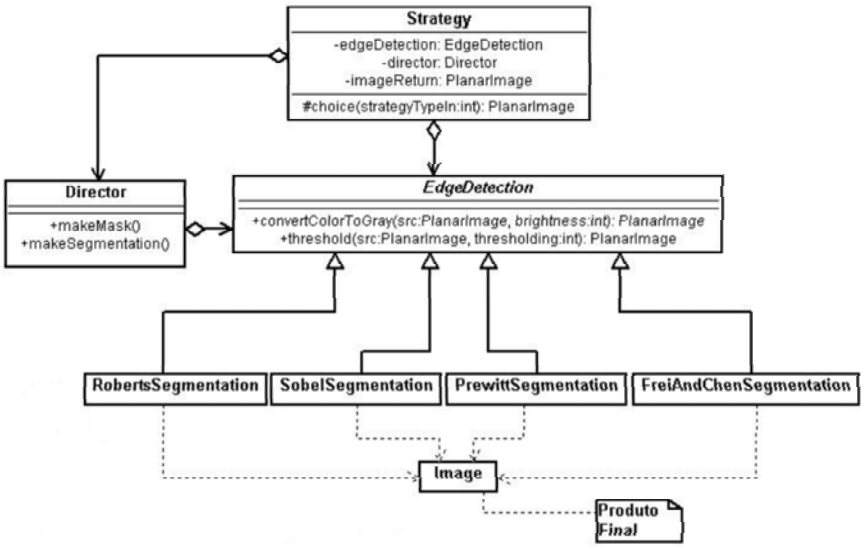


Figura 5: A visão de projeto completa do componente de segmentação baseado em máscaras de convolução.

para todas as sub-classes de um componente. Para isso, a classe mãe implementa um método (que por sua vez encapsula um algoritmo) que é utilizado por todas as sub-classes evitando assim sua replicação em cada uma delas.

Para o processo de segmentação ocorrer não basta apenas aplicar filtros de convolução. É necessário ainda, como descrito na seção 2, tratar a imagem de entrada e também complementar o processo de realce dos filtros pelo processo de binarização. Se a imagem que se está intencionando segmentar for RGB (usualmente conhecidas como multiespectral ou colorida) o componente de segmentação deve convertê-la para tons de cinza, uma vez que é mais simples operar sob esse tipo de imagem além de ser pré-requisito para o processo de limiarização futura. De outro lado, após a aplicação de determinada máscara é utilizado o processo de binarização (thresholding) para garantir a exclusão dos pixels que pertencem a borda daqueles que pertencem ao plano de fundo da imagem. Assim, cada classe concreta de `EdgeDetection` , da figura 5, deve implementar ambas funcionalidades para garantir a eficácia do componente. No entanto, como a lógica de ambas operações se mantém igual independente da classe concreta onde está implementada, é possível tratá-las de uma forma mais elegante. Para isso, basta implementar esses métodos na própria classe abstrata (usualmente chamada de mãe) e apenas chamá-los quando for conveniente nas classes filhas. Assim, evita-se replicação de código nas classes concretas (comumente chamada de filhas). Dessa forma, esses métodos se tornam modelos justificando então o seu nome pela literatura.

3.4. O padrão de controle *Command*

Esse padrão de projeto foi utilizado para construir a interface gráfica do componente de segmentação. Quando esse componente cresce em funcionalidade, como por exemplo a implementação de diversos filtros de convolução, naturalmente o usuário estará habilitado a realizar, via interface gráfica, diversas solicitações a esses filtros. Dessa forma, para gerenciar essa complexidade de requisições por parte do usuário foi utilizado o padrão de projeto *Command*.

Quando um sistema cresce muitas opções de funcionamento podem ser requisitadas via interface gráfica. Em Java, normalmente esse processo é inteiramente gerenciado pelo método `actionPerformed` e do objeto `ActionEvent` . Esse método controla as requisições feitas pelo usuário como, por exemplo, um click em um botão que será

seguido pela chamada de algum outro método. Veja na figura 6 como isso ocorre.

Código 1: Controlando requisições do usuário via Interface Gráfica.

```
01.  public void actionPerformed(ActionEvent w) {
02.     String opcao = w.getActionCommand();
03.     if (opcao.equals("Prewitt")){
04.       Prewitt();
05.     }
06.     if (opcao.equals("Roberts")){
07.       Roberts();
08.     }
09.     ...
10.  }
```

Figura 6: Implementação da gerência dos algoritmos via componentes gráficos

A palavra `Prewitt` e `Roberts` da linha 3 e 6 representam o identificador de determinados componentes gráficos especificados pelo programador. Entretanto, infelizmente, em um sistema que apresenta muitas operações esse método ficaria sobrecarregado além do que, a mesma classe onde esse método é implementado possivelmente também abrigará todos os métodos que implementam as mais distintas operações. Essa é uma forma possível e funcional, porém nada legível e portanto, de difícil manutenção. Ressalta-se ainda que subtituir a sequência de "ifs" pela sentença "switch-case" apenas tornaria o código menos intuitivo ainda pois essa última estrutura não permite controlar variáveis do tipo "Strings" mas, apenas tipos primitivos. Utilizando palavras para implementar o controle torna o código mais acessível o que não ocorre com tipos inteiros ou caracteres.

Para resolver este problema, é utilizado uma abordagem mais modular. Cada componente gráfico, que requisita alguma operação, é implementado como uma nova classe. Cada uma dessas classes herda as características de uma interface publica chamada `CommandInterFace` que é o objeto *Command* propriamente dito. A figura 7 expõe de maneira mais clara essa solução.

Na linha 6 da figura 7 pode-se ver que o método `actionPerformed` ficou reduzido a quatro linhas. Não só nesse caso ele sofreu essa redução, isto é, ele sempre manterá essa característica constante. Através da especificação da interface *Command* todas as classes criadas como a da linha 11 que tem a intenção de prover um item gráfico para realçar as bordas de uma imagem pelo filtro de Prewitt, terão em seu interior o mesmo nome do método presente em `CommandInterFace` , isto é, `ExecuteAction()` porém com diferentes implementações. Com isso é possível desacoplar o objeto que invoca a operação daquele que tem o conhecimento para executá-la [E. Gamma and Vlissides, 1995].

A principal desvantagem desse padrão é a proliferação de pequenas classes [Cooper, 1998]. Porém, essa desvantagem não representa um risco para o sistema, pelo simples fato de que o aspecto organizacional alcançado pelo padrão *Command* é muito grande. Qualquer alteração em algum componente gráfico pode ser facilmente realizada pois basta encontrar a sua classe e efetuar as mudanças necessárias sem mexer no resto do código. Por exemplo, se a classe da linha 11 necessitar mudar sua finalidade bastaria entrar dentro do seu método `ExecuteAction()` e alterar a chamada do método `Prewitt()` para outro qualquer. Para fins de simplificação dentro dessas pequenas classes foram invocados métodos locais. Entretanto, na maioria dos casos, acontece o

Código 2: Manipulando requisições do usuário de forma mais elegante.

```
01.  //especificação do objeto Command
02.  public interface CommandInterFace {
03.     public void ExecuteAction();
04.  }
05.  //invocando o objeto Command
06.  public void actionPerformed(ActionEvent e){
07.     CommandInterFace obj = (CommandInterFace)e.getSource();
08.     obj.ExecuteAction();
09.  }
10.  //Classe alusiva a um componente gráfico
11.  class ListenEdgeDetection extends JMenuItem
12.                                implements CommandInterFace{
13.      public void ExecuteAction(){
14.           Prewitt();
15.     }
16.  }
17.  ...
```

Figura 7: Uma alternativa de controle mais legível.

instanciamento dos objetos principais dos componentes que estão alocados em unidades do tipo pacote. A figura 8 mostra a interface gráfica desse componente.

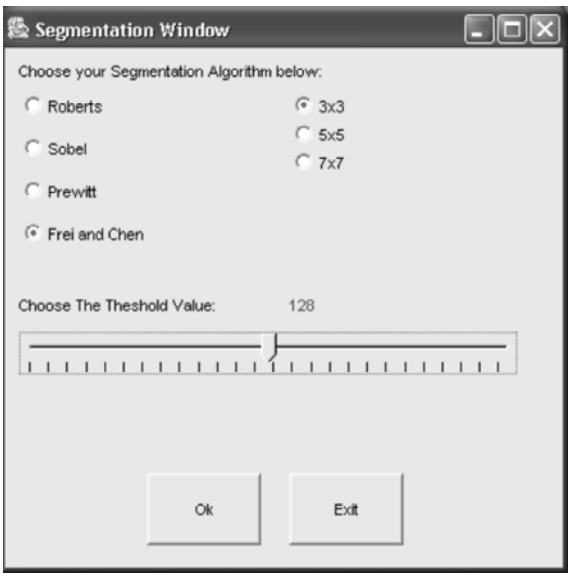


Figura 8: A interface do componente de segmentação.

4. Conclusão

Muitas operações em processamento de imagens se baseiam na aplicação de filtros [Akre and Tabirca, 2003]. Dessa forma, a utilização desses dois padrões, especialmente pelo *Builder* que dá o suporte central ao componente, podem ser visto com muita utilidade no desenvolvimento de software para imagens. Um bom exemplo é a utilização desses padrões para construir um componente para morfologia binária, onde o elemento

estruturante apresenta diferentes formatos e tamanhos. Assim pode-se utilizar o padrão *Builder* para construir passo-a-passo toda a representação necessária para implementar um objeto que representa o elemento estruturante para, posteriormente utilizá-lo em operações de dilatação, erosão, abertura, fechamento e outras. Em nível de aplicação esse componente se segmentação baseado em contornos pode ser visto como um pré-requisito para um componente de morfologia binária pois, o alvo de estudo dessa parte de morfologia são imagens binárias que por sua vez são o resultado final desse componente de segmentação. Assim, também a nível de funcionalidade um componente acaba ajudando o outro na resolução de algum problema.

O padrão *Command* propiciou regras de controle para o gerenciamento de toda a interface gráfica do componente de segmentação. Qualquer modificação ou agregação de novos elementos gráficos basta seguir um modelo onde todos esses elementos são considerados como classes isoladas que contém operações comuns porém com diferentes implementações. Assim, quando o componente for se tornando grande e, conseqüentemente a sua interface gráfica também, a principal vantagem oferecida pelo padrão *Command* é a forma com que ele gerencia a complexidade resultante da associação entre componentes gráficos e requisição dos algoritmos dos filtros de convolução.

Finalizando, a modelagem do componente através da notação UML, propicia ao desenvolvedor a visão necessária para trabalhar a parte mais difícil na concepção de um sistema. Essa parte é o processo de reconhecer e enxergar a aplicação de um determinado padrão para um problema de imagens. Esse problema no início é muito grande, principalmente no momento de conjugar os vários padrões identificados que compõem o componente. E para esse processo de reconhecimento, é necessário ter conhecimento de todos os padrões apontados pela literatura e, também, conhecimento sobre o problema na área de imagens. Se esses dois aspectos estão dissociados não é possível efetuar uma implementação que assegure características como reuso e legibilidade de código-fonte.

Referências

Akre, A. and Tabirca, S. (2003). Imaging technologies in java. pages 159–161, Kilkenny City, Ireland. Proceedings of the 2nd international conference on Principles and practice of programming in Java.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern - Oriented Software Architecture: A System of Patterns*, volume 1. John Wiley & Sons Ltd, New York.

Cooper, J. W. (1998). *The Design Patterns - Java Companion*. Addison-Wesley.

E. Gamma, R. Helm, R. J. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, New York.

Gonzalez, R. C. and Woods, R. E. (1992). *Digital Image Processing*. Addison-Wesley.

Gosling, J. (2004). The mars mission continues. Disponível em: <`http://www.sun.com/mars`>. Acesso em: abr. 2004. Sun Microsystems, Inc.

Metsker, S. J. (2002). *Design Patterns Java Workbook*. Addison-Wesley.

Ritter, G. X. and Wilson, J. N. (2000). *Handbook of Computer Vision Algorithms in Image Algebra*. CRC Press, New York.

Russ, J. C. (1998). *The Image Processing Handbook*. CRC Press.

Aplicação de metapadrões e padrões em desenvolvimento de software para sistemas de informação

Gabriela T. de Souza[1, 2], Carlo Giovano S. Pires[1], Fabiana Gomes Marinho[1], Arnaldo Dias Belchior[2]

[1]Instituto Atlântico
Rua Chico Lemos, 946 – 60 822-780 – Fortaleza – CE – Brasil

[2]Universidade de Fortaleza
Av. Washington Soares, 1321 – Fortaleza – CE – Brasil

{gabi, cgiovano, fabiana}@atlantico.com.br, belchior@unifor.br

***Abstract**. This work presents a catalogue of metapatterns and their application in the development of information systems. The considered metapatterns are implemented by requirement patterns, design patterns and test patterns. RUP is used as reference of software engineering good practices.*

***Resumo.** Este trabalho apresenta um catálogo de metapadrões e a aplicação desses metapadrões no desenvolvimento de sistemas de informação. Os metapadrões propostos são implementados por padrões de requisitos, padrões de projeto e padrões de teste. O RUP foi utilizado como referência de boas práticas de engenharia de software a serem seguidas.*

1. Introdução

Em sistemas de informação, as funcionalidades são muito concentradas em cenários de uso baseados em manutenção e consulta de dados, gerando necessidades e problemas recorrentes durante o desenvolvimento do sistema, que afetam os vários produtos de trabalho ao longo do ciclo de vida de desenvolvimento de software. Entre estes problemas podemos citar o tratamento de operações para criação, atualização, exclusão e consulta de entidades.

Com o objetivo de minimizar estes problemas, vários modelos e processos foram propostos para auxiliar o desenvolvimento de produtos de software de qualidade, destacando-se o RUP (Rational Unified Process) [8]. O RUP é um *framework* de processo adaptável que abrange as melhores práticas do desenvolvimento de software de mercado e tem sido largamente utilizado em projetos de software.

Desenvolver software usando padrões pode reduzir o custo e condensar o ciclo de vida do desenvolvimento, e simultaneamente manter a qualidade dos sistemas desenvolvidos. Entretanto, o potencial de usar padrões em sistemas de software não é aproveitado inteiramente. Embora diversos padrões tenham sido desenvolvidos para analisar, projetar, e implementar o software; não existe nenhuma orientação ou metodologia madura que fornece uma abordagem sistemática para integrar estes diferentes tipos de padrões durante o ciclo de desenvolvimento.

Neste trabalho, propomos o uso de um catálogo de metapadrões integrado ao ciclo de vida de desenvolvimento de software para apoiar a construção de sistemas de informação. Os metapadrões apresentados são implementados por padrões de requisitos,

padrões de projeto e padrões de teste. Não abordamos padrões de implementação. Esses padrões serão tratados em trabalhos futuros.

Este trabalho está organizado em 5 seções. A seção 2 descreve modelos e processos de software, que são propostos para auxiliar no desenvolvimento de software. Os conceitos de padrões e metapadrões são apresentados na seção 3. Na seção 4, descrevemos o catálogo de metapadrões proposto. A seção 5 apresenta o uso de metapadrões e padrões no desenvolvimento de sistemas de informação. Finalmente, a seção 6 contém as conclusões e os direcionamentos para trabalhos futuros.

2. Desenvolvimento de software

Atualmente, existem vários processos de desenvolvimento software com o objetivo de auxiliar os grupos de desenvolvimento a construírem produtos de software de qualidade, capazes de atender as necessidades e exigências dos usuários. Apesar de possuírem abordagens diferentes, as disciplinas descritas a seguir são comuns a vários processos [14].

- Requisitos: define as funcionalidades e as restrições do software.
- Análise e Projeto: produz a arquitetura utilizada como base para o desenvolvimento do software.
- Implementação: produz e libera o código para o cliente final.
- Testes: valida o software para garantir que as funcionalidades e restrições serão atendidas.
- Manutenção: garante que o software atenda às necessidades de mudança do cliente.

Estas disciplinas fundamentais são organizadas de acordo com os modelos de ciclo de vida. O ciclo de vida cascata (clássico) é o mais tradicional. Neste ciclo de vida o desenvolvimento de software é organizado de forma a percorrer cada disciplina em seqüência apenas uma vez. O ciclo de vida iterativo é uma alternativa mais flexível para o desenvolvimento de software. As diversas disciplinas são percorridas várias vezes, gerando um melhor entendimento dos requisitos, planejando uma arquitetura robusta, elevando a organização do desenvolvimento e, por fim, liberando uma série de implementações que são gradualmente mais completas. No ciclo de vida incremental as necessidades do usuário são determinadas e os requisitos do sistema são definidos e, em seguida, o restante do desenvolvimento é realizado em uma seqüência de incrementos. O primeiro incremento incorpora partes das capacidades planejadas, o próximo incremento adiciona mais capacidades e assim por diante até o sistema estar completo [13].

Na prática, abordagens híbridas dos ciclos de vida descritos podem ser utilizadas. O RUP, por exemplo, é um *framework* de processo iterativo e incremental que provê uma abordagem disciplinada para o desenvolvimento de software [8].

Conforme apresentado na Figura 1, o RUP possui duas dimensões. O eixo horizontal representa o aspecto dinâmico do processo e mostra as fases do ciclo de vida à medida que este se desenvolve. O eixo vertical representa o aspecto estático do processo, como ele é descrito em termos de disciplinas [10].

As disciplinas fundamentais do processo de desenvolvimento de software também estão presentes na estrutura do RUP. A disciplina de *Requisitos* é responsável por estabelecer e manter concordância com os clientes e outros envolvidos sobre o que o sistema deve fazer, oferecer aos desenvolvedores do sistema uma compreensão melhor dos requisitos e definir as fronteiras do sistema. O RUP indica a utilização de casos de uso para definir, detalhar e documentar requisitos. Um caso de uso define um conjunto de instâncias de casos de uso, no qual cada instância é uma seqüência de ações realizada por um sistema que produz um resultado de valor observável para determinado ator [10].

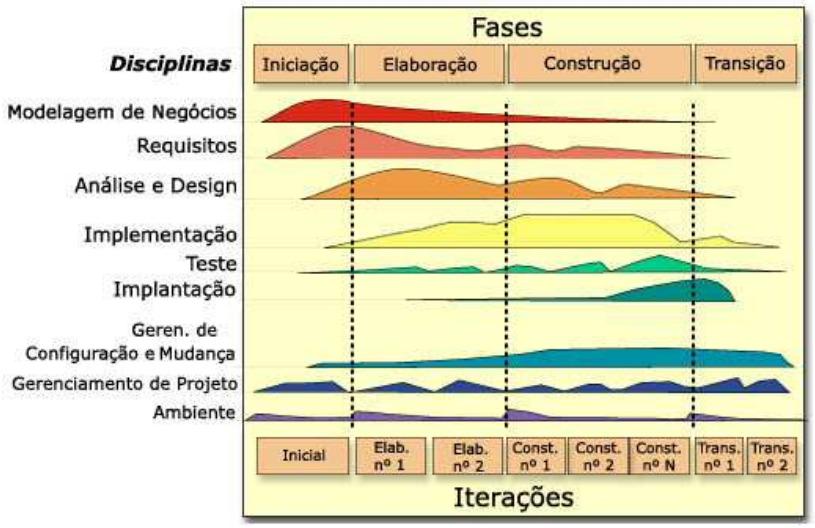


Figura 1: Ciclo de vida de desenvolvimento do RUP

A disciplina de *Análise e Projeto*, por sua vez, visa transformar os requisitos em um projeto do sistema e desenvolver a arquitetura. O processo é baseado em caso de uso e desenvolve a análise e projeto através de Realizações de Casos de Uso. A finalidade da disciplina de *Implementação* é implementar classes e objetos, testar e integrar os resultados produzidos. A disciplina de *Testes* atua em vários aspectos como uma provedora de serviços para as outras disciplinas, enfatizando principalmente a avaliação da qualidade do produto.

Neste trabalho, propomos um conjunto de metapadrões e padrões para apoiar o processo de desenvolvimento de sistemas de informação baseado no RUP. Os metapadrões e padrões propostos podem ser aplicados ao longo de todo o ciclo de vida de desenvolvimento de software com suas fases e disciplinas de forma integrada e consistente. Apesar de utilizar o RUP como base para o ciclo de vida de desenvolvimento de software, a abordagem pode ser facilmente aplicada a outros processos, dada a existência das disciplinas fundamentais.

3. Padrões de software

Um padrão é definido em [1] como uma regra que expressa uma relação entre um determinado contexto, um problema e uma solução. Este conceito tem sido amplamente utilizado no domínio da engenharia de software como uma forma de descrever boas soluções para problemas específicos em todo o ciclo de vida do projeto [2].

Uma classificação bastante utilizada para padrões de software toma como base o estágio de desenvolvimento de software em que o padrão é aplicado. Em [2], os padrões

são classificados em cinco categorias: padrões de requisitos, padrões de análise, padrões de projeto, idiomas e padrões de testes. Os padrões de testes são orientações para as atividades de testes, incluindo documentação, execução e divulgação dos resultados [11]. Os idiomas são orientações para codificar padrões de projeto em uma linguagem de programação específica. Os padrões de projeto são utilizados para refinar os componentes ou relacionamentos entre eles podendo ser usados durante toda a fase de projeto do software. O objetivo dos padrões de análise é construir um modelo de análise que represente as estruturas conceituais dos processos do negócio. Os padrões de requisitos, por sua vez, documentam as necessidades dos usuários e o comportamento dos sistemas em um alto nível de abstração.

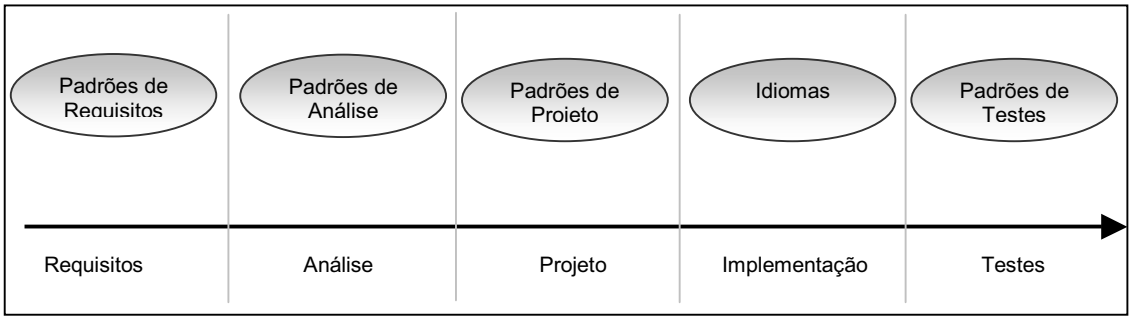


Figura 2: Classificação pelo ciclo de vida de desenvolvimento (adaptado de [12])

Metapadrões representam uma abordagem proposta por [9], que consiste na especificação de um conjunto de metapadrões que descrevem como construir *frameworks*. Segundo [9], metapadrões constituem uma abordagem elegante e poderosa que pode ser aplicada para classificar e descrever padrões em um metanível. Portanto, metapadrões não substituem as abordagens de padrões, mas complementam-nas.

O conceito de linguagens de padrões foi introduzido por [1]. Em uma adaptação para a engenharia de software, Coplien descreveu uma linguagem de padrões como uma coleção de padrões que trabalham em conjunto para construir um sistema [4]. As linguagens de padrões, além dos padrões que as compõem, possuem um título, geralmente possuem uma descrição ou resumo e um mapa, que consiste de um grafo que ilustra como seus padrões estão relacionados [12]. Segundo [7], uma linguagem de padrões deve ser completa morfologicamente e completa funcionalmente. Se os padrões não são completos destas duas maneiras, então eles são considerados uma simples coleção de padrões ou catálogo de padrões, como [6] [5] [3].

4. Catálogo de Metapadrões

Nesta seção, apresentamos uma breve descrição dos metapadrões e padrões que compõem o catálogo de metapadrões proposto para auxiliar o processo de desenvolvimento de sistemas de informação. Este trabalho aplica e amplia o conceito de metapadrões proposto por PREE [9] no contexto de padrões de projeto, para metapadrões no domínio de sistemas de informação. O foco deste trabalho está nos padrões de requisitos, padrões de projeto e padrões de teste. Os padrões de implementação serão abordados em trabalhos futuros.

A Figura 3 apresenta os relacionamento entre os metapadrões e padrões. Os padrões citados nesta seção encontram-se descritos em [15] [16].

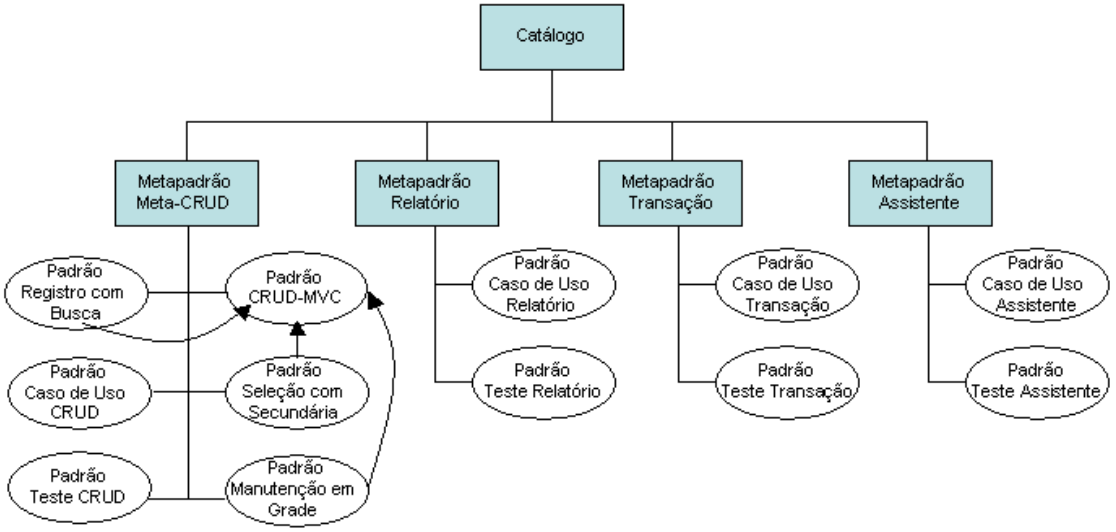


Figura 3: Relacionamento entre os metapadrões e os padrões

4.1. Metapadrão Meta-CRUD

Problema

Tratar a manutenção das entidades nas diversas fases de um ciclo de vida de construção de software de forma integrada e consistente.

Solução

Este metapadrão descreve a estrutura geral para manutenção de entidades. As entidades devem ser mantidas através de operações de criação, consulta, alteração e consulta (CRUD – *Create, Read, Update, Delete*). A estrutura dessas operações deve considerar o tipo de entidade, sua complexidade e volume de dados tratados.

Padrões que implementam o metapadrão

- Caso de Uso CRUD;
- CRUD-MVC;
- Registro com Busca;
- Manutenção em Grade;
- Seleção com Secundária;
- Teste CRUD.

4.1.1. Padrão Caso de Uso CRUD

Contexto

Este padrão é utilizado para a documentação dos requisitos de manutenção em sistemas da informação, por meio do uso de modelos e especificações de casos de uso. Os requisitos de manutenção são caracterizados por operações de Inclusão, Consulta, Alteração e Exclusão.

Problema

Como documentar os requisitos funcionais de inserção, atualização, exclusão e consulta de dados por meio de especificações de casos de uso?

4.1.2. Padrão CRUD-MVC

Contexto

Sistemas de informação requerem funcionalidades de negócio implementadas através de interface humano-computador (IHC), componentes para tratamento de regras de negócio e acesso a dados para operações CRUD e operações de negócio.

Problema

Como tratar funcionalidades recorrentes de criação, consulta, atualização e exclusão de dados em sistemas de informação considerando aspectos de apresentação e tratamento de eventos, regras de negócio e persistência?

4.1.3. Padrão Registro com Busca

Contexto

Utilizado para as principais entidades de negócio, com muitos campos e/ou relacionamentos. Em geral, essas entidades são objetos complexos com muitos atributos e relacionamentos. O enfoque do cenário é de uma busca eficiente seguida de visualização e edição de uma entidade.

Problema

Como criar componentes de cadastro e manutenção de entidades de negócio complexas, atendendo a requisitos de interface humano-computador, permitindo a reutilização de interação com usuário e estrutura comuns aos vários tipos de entidades complexas existentes em uma aplicação de sistema de informação?

4.1.4. Padrão Manutenção em Grade

Contexto

Sistemas de informação utilizam entidades básicas e simples para configuração do sistema. Essas entidades possuem poucos atributos, sem objetos dependentes e possuem um pequeno número de instâncias, por exemplo, cadastro de unidades federativas, tipos de endereço, tipo de cliente, entre outras. Entidades básicas são usadas em relacionamentos com entidades de negócio, por exemplo, no momento de cadastrar um cliente, é necessário informar a unidade federativa de seu endereço, o tipo de endereço e o tipo de cliente. A implementação de entidades básicas em sistemas de informação não alcança, de forma trivial, um bom grau de reuso e eficiência.

Problema

Como implementar funcionalidades de cadastro de entidades básicas de forma eficiente e reutilizável?

4.1.5. Padrão Seleção com Secundária

Contexto

Utilizado para entidades principais ou secundárias de média ou alta complexidade com vários atributos e relacionamentos em situações que requerem a seleção de entidades sob determinado critério para somente depois realizar edição ou inclusão em objeto de interface auxiliar, e não diretamente sobre a grade. A edição em objeto auxiliar (interface secundária) facilita a edição de um número maior de atributos e relacionamentos.

Problema

Como implementar funcionalidades de seleção e manutenção de entidades de negócio de média e alta complexidade de forma eficiente e reutilizável?

4.1.6. Padrão Teste CRUD

Contexto

Este padrão é utilizado para especificar os casos de teste das operações de Inclusão, Consulta, Alteração e Exclusão. O padrão indica idéias de testes típicas e cenários de falhas recorrentes para inserção, atualização, exclusão e consulta de dados em sistemas de informação, de forma a facilitar e agilizar a execução dos testes.

Exemplos de idéias de testes típicas para inserção: a) Inserir entidade já existe e verificar resultado b) Inserir entidade não existe e consultar em seguida para verificar se os dados estão iguais aos dados solicitados na inserção.

Problema

Como especificar os casos de teste dos requisitos funcionais de inserção, atualização, exclusão e consulta de dados por meio de especificações de testes?

4.2. Metapadrão Relatório

Problema

Tratar a geração de relatórios nas diversas fases do ciclo de vida de desenvolvimento de software de forma integrada e consistente.

Solução

Este metapadrão descreve a estrutura geral para geração de relatórios. O relatório deve permitir a parametrização (filtros), visualização e exportação de dados. Estruturas complementares como agrupamento e totalizações são fornecidas.

Padrões que implementam o metapadrão

- Caso de Uso Relatório;
- Teste Relatório.

4.2.1. Padrão Caso de Uso Relatório

Contexto

Em sistemas de informação, uma grande quantidade de dados é armazenada freqüentemente. Neste contexto, surge a necessidade de visualizar, exportar ou imprimir dados armazenados com o objetivo de conferir, analisar e tomar decisões com base nesses dados.

Problema

Como documentar os requisitos de relatórios que podem incluir a necessidade de visualizar, exportar ou imprimir dados de entidades de acordo com filtros especificados, agrupamentos, totalizações e informações a serem apresentadas?

4.2.2. Padrão Teste Relatório

Contexto

Este padrão é utilizado para especificar os casos de teste de relatórios. O padrão apresenta idéias de testes típicas e cenários de falhas recorrentes na visualização, exportação ou impressão dados de entidades de acordo com filtros especificados, agrupamentos, totalizações e informações a serem apresentadas.

Exemplos de idéias de testes típicas para relatório: a) Verificar resultados em combinações de filtros de relatório b) Verificar visualização e impressão c) Verificar totalizações, cálculos e agrupamentos d) Verificar formato e) Verificar formato e dados em arquivos exportados.

Problema

Como especificar os casos de teste de relatórios?

4.3. Metapadrão Assistente

Problema

Tratar o processamento de operações complexas, com necessidade de iteração com usuários, nas diversas fases de um ciclo de vida de desenvolvimento de software de forma integrada e consistente.

Solução

Este metapadrão descreve a estrutura geral para organizar operações baseadas em assistentes. O assistente deve organizar a operação em passos, de forma que cada passo tenha início em um ponto onde necessite configurações ou decisões do usuário.

Padrões que implementam o metapadrão

- Caso de Uso Assistente;
- Teste Assistente.

4.3.1. Padrão Caso de Uso Assistente

Contexto

Este padrão é utilizado para a documentação dos requisitos de operações complexas que são executadas em diversos passos, onde decisões ou dados necessitam serem informados em cada passo através da iteração com o usuário.

Problema

Como documentar os requisitos de uma operação, na qual diversas decisões devem ser tomadas antes que a operação possa ser concluída completamente?

4.3.2. Padrão Teste Assistente

Contexto

Este padrão é utilizado para especificar os casos de teste de operações complexas, na qual diversas decisões devem ser tomadas antes que a operação possa ser concluída completamente. O padrão indica idéias de testes típicas e cenários de falhas recorrentes na seqüência de passos, retorno, parametrização e decisões.

Exemplos de idéias de testes típicas para assistente: a) Verificar seqüência correta dos passos b) verificar mensagens e explicações dos passos; c) Verificar se as informações dos passos anteriores são passadas de forma correta entre os contextos d) Verificar resultado final da transação.

Problema

Como especificar os casos de teste de uma operação, na qual diversas decisões devem ser tomadas antes que a operação possa ser concluída completamente?

4.4. Metapadrão Transação

Problema

Tratar operações longas e complexas no formato de comandos e compostas por um conjunto de transações nas diversas fases de um ciclo de vida de construção de software de forma integrada e consistente.

Solução

Este metapadrão descreve a estrutura geral para organizar operações de comando. A operação deve ser disparada por uma parametrização, manter o usuário informado da evolução das transações e tratar a consistência entre as transações.

Padrões que implementam o metapadrão

- Caso de Uso Transação;
- Teste Transação.

4.4.1. Padrão Caso de Uso Transação

Contexto

Este padrão é utilizado para a documentação dos requisitos de operações que são tratadas como um comando atômico que processa várias transações. Tipicamente operações *batch* e operações que requerem apenas um comando de inicio do caso de uso pelo usuário tendo pouca entrada de dados e iteração com o sistema.

Problema

Como documentar os requisitos de operações que possuem a execução de longa duração ou que são executadas em formato de comando atômico, dando ênfase para os requisitos especiais dessas operações?

4.4.2. Padrão Teste Transação

Contexto

Este padrão é utilizado para especificar os casos de teste de operações que possuem execução de longa duração ou que são executadas em formato de comando atômico. O padrão indica idéias de testes típicas e cenários de falhas recorrentes na parametrização, no tratamento de falhas e na recuperação de transações.

Exemplos de idéias de testes típicas para transação: a) Verificar se parametrização da transação é requerida de acordo com especificação b) Solicitar execução da transação sem informar os parâmetros necessários c) Verificar resultado final da transação de acordo com a parametrização; d) Executar da transação e provocar

falha do durante processamento (interromper aplicação por exemplo) e verificar a consistência do resultado.

Problema

Como especificar os casos de teste de operações que possuem execução de longa duração ou que são executadas em formato de comando atômico?

5. Desenvolvimento de um sistema de informações usando metapadrões

Os principais produtos de trabalho no desenvolvimento de um sistema de informação são gerados nas disciplinas de Requisitos, Análise e Projeto, Implementação e Teste. Para direcionar a solução no contexto de produto de trabalho, propomos a utilização de metapadrões, que aplicados a cada disciplina, definem um conjunto de padrões que podem ser aplicados.

A Figura 4 apresenta os metapadrões e padrões propostos e seus relacionamentos de acordo com as principais disciplinas do ciclo de vida de desenvolvimento. Cada um dos padrões detalha como aplicar uma solução geral indicada nos metapadrões. Vale a pena ressaltar que os padrões fornecem a solução para a elaboração dos produtos de trabalho.

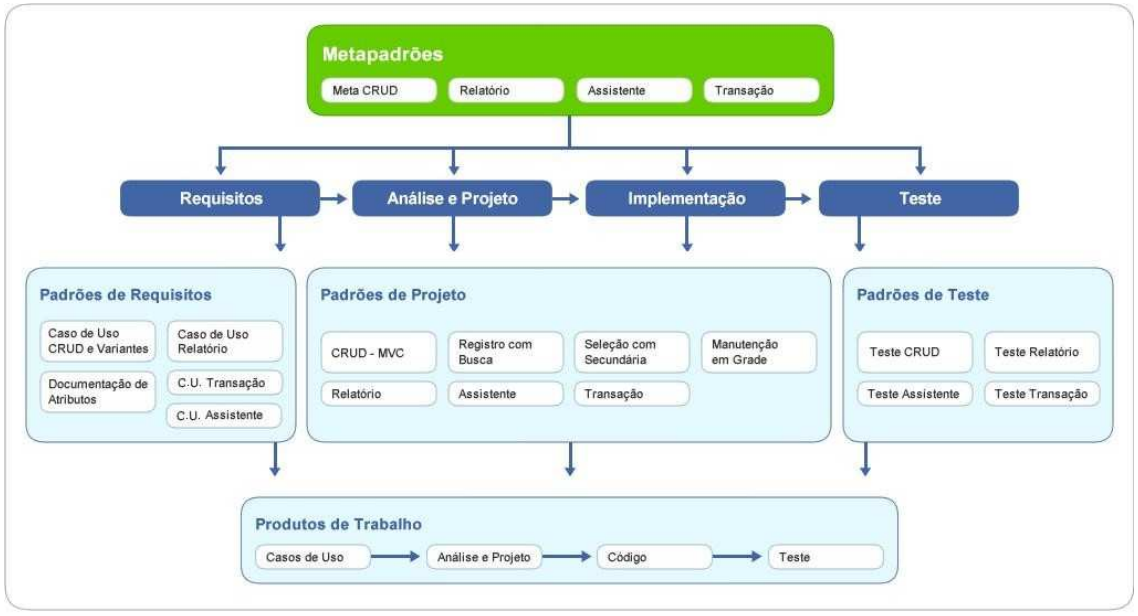


Figura 4: Relacionamento entre os metapadrões e os padrões no ciclo de desenvolvimento

A aplicação de padrões poderá auxiliar na construção de produtos de trabalho de acordo com o ciclo de desenvolvimento de forma mais consistente, dado que os padrões são delineados com base nos metapadrões para desenvolvimento de sistemas de informação.

Nas fases iniciais, os metapadrões associados aos diversos requisitos (casos de uso) do sistema, são identificados. À medida que o ciclo de vida evolui, os produtos de trabalho derivados são desenvolvidos tomando como base o produto gerado na disciplina anterior. Em paralelo, os padrões são selecionados de acordo com o metapadrão associado ao caso de uso e a disciplina em execução.

Nas fases iniciais de desenvolvimento, analistas de requisitos especificam casos de uso com base nos padrões de requisitos. Esses casos de uso servem de base para a análise e projeto. Além dos próprios casos de uso, o projetista terá como insumo os padrões de projeto. Esses padrões possuem forte consistência com a estrutura dos casos de uso, pois são baseados nos mesmos metapadrões, facilitando o trabalho do projetista. Da mesma forma, os casos de testes são construídos com base nos casos de uso e padrões de testes de forma consistente e facilitada. De acordo com o RUP, o código é mapeado e criado com base na Análise e Projeto. Assim, o código criado estará consistente com os padrões utilizados na Análise e Projeto.

Por exemplo, quando um caso de uso é associado ao metapadrão *Meta-CRUD*, durante a execução das atividades de especificação de casos de uso da disciplina de *Requisitos*, o padrão *Caso de Uso CRUD* é selecionado para o detalhamento do caso de uso. Em seguida, durante a *Análise e Projeto* do caso de uso, os padrões de projeto associados ao metapadrão *Meta-CRUD* são verificados para uso na criação do projeto do caso de uso. O padrão é então selecionado com base na adequação do padrão ao contexto e problema. Esse processo é repetido até se chegar na seleção do padrão *Teste CRUD* para criação da especificação de testes (ver Figura 5).

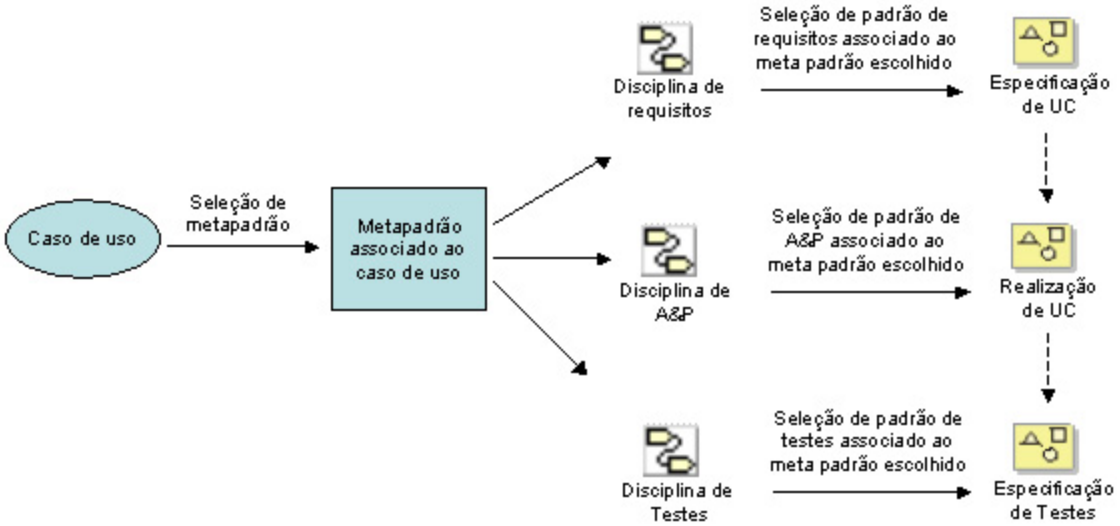


Figura 5: Aplicação de metapadrões e padrões no ciclo de desenvolvimento

6. Conclusões

Este trabalho apresentou um catálogo de metapadrões e uma abordagem para definição e uso dos padrões ao longo do ciclo de desenvolvimento de software em sistemas de informação. Os metapadrões propostos solucionam problemas genéricos no contexto de sistemas de informação a partir do uso de padrões. O trabalho abordou metapadrões, padrões de requisitos, padrões de projeto e padrões de teste.

O RUP foi utilizado como referência de boas práticas de engenharia de software a serem seguidas e como base para o ciclo de vida de desenvolvimento de software.

De forma geral, o trabalho forneceu:

- Uma abordagem para a aplicação de metapadrões e padrões durante o ciclo de desenvolvimento de sistemas de informação.

- Metapadrões que organizam padrões de forma integrada e consistente para as diversas disciplinas de construção de software.
- Uma abordagem integrada das disciplinas, do modelo de ciclo de vida iterativo e de produtos de trabalho do RUP.

Como trabalhos futuros, temos a pesquisa de idiomas derivados de metapadrões propostos para linguagens como Java. Além disso, o desenvolvimento de um *framework* que suporte a aplicação dos metapadrões em linguagens de desenvolvimento pode facilitar a consistência do código com a análise e projeto e aumentar a produtividade.

Referências

[1] ALEXANDER, C. et al. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York, NY, 1977.

[2] ANDRADE, R. Capture, Reuse, and Validation of Requirements and Analysis Patterns for Mobile Systems. Ph.D. Thesis, School of Information Technology and Engineering (SITE), University of Ottawa, Ottawa, Ontario, Canada, May 2001.

[3] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. Pattern-Oriented Software Architecture. John Wiley and Sons, New York, NY, 1996.

[4] COPLIEN, J. O. Software Patterns. SIGS books and Multimedia, June 1996.

[5] Core J2EE Pattern Catalog. Disponível em: http://java.sun.com/blueprints/corej2eepatterns. Acessado em: 06/04/2005.

[6] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[7] HANMER, R. Introduction to Pattern Languages. SugarLoafPLoP 2003, The Third Latin American Conference on Pattern Languages of Programming, Porto de Galinhas, PE, 2003.

[8] POLLICE, Gary. Using the Rational Unified Process for Small Projects: Expanding Upon Extreme Programming. Rational Software White Paper.

[9] PREE, W. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

[10] RATIONAL UNIFIED PROCESS Tutorial. Versão 2002 05 00.

[11] RISING, L. The Pattern Almanac 2000. Software Pattern Series, Addison-Wesley, 2000. ISBN 0-201-61567-3.

[12] SANTOS, M. S. Uma Proposta para a Integração de Modelos de Padrões de Software com Ferramentas de Apoio ao Desenvolvimento de Sistemas. Dissertação de Mestrado. Universidade Federal do Ceará, Fortaleza, 2004.

[13] Software Development and Documentation, MIL-STD-498, Departamento de Defesa dos EUA, dezembro de 1994.

[14] SOMMERVILLE, I. **Software Engineering**. 6th Edition, Addison-Wesley Publishers Ltd., 2001. ISBN 0-201-39815-X.

[15] SOUZA, G. T. e PIRES, C. G. PATI-MVC: Uma Família de Padrões para Sistemas de Informação Baseada no Padrão MVC. SugarloafPloP, 2004.

[16] SOUZA, G. T., PIRES, C. G. e Barros, M. Padrões MVC para Sistemas de Informação. SugarloafPloP, 2003.

Relacionamento de Padrões de Engenharia de Software e de Interação Humano-Computador para o Desenvolvimento de Sistemas Interativos

André Constantino da Silva [1,†], Júnia Coutinho Anacleto Silva[1], Rosângela Aparecida Dellosso Penteado[1], Sérgio Roberto Pereira da Silva[2]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – CEP 12.565-905 – São Carlos – SP – Brazil

[2]Departamento de Informática – Universidade Estadual de Maringá (UEM)
Av. Colombo, 5790, Zona 07 – CEP 87.020-900 – Maringá – PR – Brazil

{andrecons,junia,rosangel}@dc.ufscar.br, srsilva@din.uem.br

Abstract. *Many Software Engineering and Human-Computer Interaction patterns have been identified and published lately. However, as these knowledge areas are complementary in the interactive system development, there is a need for researches that consider the unified use of those kinds of patterns in a development process. In this context, this paper presents a case study developed in order to demonstrate and detail the application of SE and HCI patterns in an interactive system development process. As a result, it was identified nineteen relationships among the applied patterns.*

Resumo. *Ultimamente muitos padrões tanto de Engenharia de Software quanto de IHC são identificados e divulgados. Entretanto, tendo em vista o fato dessas áreas de conhecimento serem complementares no desenvolvimento de sistemas interativos, há uma carência de pesquisas que abordam a aplicação conjunta dos padrões dessas áreas durante o processo de desenvolvimento. Assim, este artigo apresenta um estudo de caso desenvolvido com o objetivo de abordar e detalhar a aplicação de padrões de ES e de IHC no processo de desenvolvimento de sistemas interativos. Como resultados, foram identificados dezenove relacionamentos entre os padrões aplicados.*

1. Introdução

Este artigo, motivado pelo fato de padrões serem identificados e aplicados pelas áreas de Engenharia de Software (ES) e de Interação Humano-Computador (IHC) durante o desenvolvimento de sistemas interativos, tem por objetivo apresentar um conjunto de relacionamentos identificados a partir da aplicação conjunta de padrões de ES e de IHC em um processo de desenvolvimento de sistemas interativos baseado no modelo de processo Prototipação.

†Bolsista financiado pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

Sabendo que existe um relacionamento entre as áreas de ES e de IHC para desenvolver sistemas interativos de forma mais abrangente, acredita-se que tal relacionamento também pode ser expresso através de relacionamentos entre padrões. Devido a pouca preocupação dos escritos em identificar relacionamentos entre padrões de outras áreas, tais relacionamentos podem ser coletados durante a aplicação de padrões na elaboração de sistemas em um processo de desenvolvimento que considere as visões das áreas de IHC e de ES.

Contextualizando, sabe-se que o desenvolvimento de um sistema interativo é um processo complexo, com diversas preocupações a serem consideradas, tais como: a organização do processo de desenvolvimento e da equipe que realizará as atividades propostas no processo. Durante a organização de tal processo, devem-se considerar atividades que englobem a engenharia de requisitos (considerando o levantamento, especificação, análise e verificação dos requisitos do sistema), o projeto, a implementação, e a validação e verificação do sistema. Devido à natureza de um sistema interativo, também é necessário considerar atividades que estão relacionadas à elaboração, avaliação e refinamento de protótipos.

Diversos padrões de ES e de IHC podem ser aplicados durante a realização das atividades de um processo de desenvolvimento. Entretanto, uma aplicação de padrões mais sistemática, envolvendo as visões de ambas as áreas no processo de desenvolvimento ainda não é muito divulgada, apresentando poucos artigos na literatura especializada. Devido a essa carência, muitas vezes, os profissionais se limitam a aplicar só alguns padrões, impedindo um melhor aproveitamento do potencial dos padrões e a obtenção de um produto melhor. Relacionar padrões permite que esforços de estudo de padrões sejam minimizados, pois o relacionamento indicará quais os possíveis padrões a serem aplicados em seguida, que é uma das motivações deste trabalho.

Portanto, padrões existentes na literatura que podem ser aplicados para auxiliar na realização dessas atividades foram levantados, estudados e agrupados em categorias, que são apresentadas na Seção 2. Em seguida as categorias foram relacionadas às fases do modelo de processo Prototipação, realizando estudos de caso para avaliar a aplicação conjunta dos padrões selecionados, apresentado na Seção 3. A partir da aplicação conjunta, foram obtidos alguns relacionamentos entre os padrões de ES e de IHC, que são discutidos na Seção 4. Na Seção 5 são comentados as considerações finais e os trabalhos futuros.

2. Padrões de ES e Padrões de IHC

Diversos padrões de ES e de IHC encontrados na literatura foram estudados e analisados. Para facilitar o estudo e aplicação desses padrões em um processo de desenvolvimento, eles foram agrupados em categorias. Para os padrões de ES, estendem-se aqui as categorias propostas por Buschmann *et al*. (1996), incluindo as categorias padrões de processo, padrões organizacionais, padrões de análise, padrões de persistência de dados e padrões de testes:

- **Padrões de Processo**: conduzem o desenvolvimento de software, descrevendo uma abordagem ou série de ações provadas e de sucesso para o desenvolvimento de software [Ambler, 1998]. Os padrões de processo que foram estudados e aplicados

foram os que compõem a linguagem de padrões de Coplien [Coplien, 1995], a linguagem de padrões *Requirements-Analysis-Process Pattern Language - RAPPeL* [Whitenack, 1995], a linguagem de padrões *Caterpillar's Fate* [Kerth, 1995] e a linguagem de padrões para desenvolvimento de protótipos conceituais efetivos [Stimmel, 1999];

•**Padrões Organizacionais**: auxiliam o gerenciamento das pessoas envolvidas com o processo de software [Ambler, 1998]. Exemplo de padrões organizacionais são os propostos por Coplien em sua linguagem de padrões [Coplien, 1995];

•**Padrões de Análise**: expressam grupos de conceitos que representam uma construção comum na modelagem de negócio. Eles podem ser relevantes para um domínio, ou para vários domínios [Fowler, 1996]. Para a realização deste trabalho foram considerados os padrões de análise definidos por Fowler (1996) e a linguagem de padrões para Gerência de Recurso de Negócios (GRN) [Braga *et al.,* 1999];

•**Padrões Arquiteturais**: expressam uma organização estrutural ou esquemas para sistemas. Pode-se citar, como exemplo, o *Model-View-Controller* (MVC) e o *Presentation-Abstraction-Control* (PAC) [Buschmann *et al.*, 1996] para sistemas interativos;

•**Padrões de Projeto**: refinam subsistemas ou componentes de um sistema, ou a relação entre eles. Nessa categoria têm-se os padrões identificados por Gamma *et al*. (1995) e os por Grand (1998);

•**Padrões de Persistência de Dados**: descrevem mecanismos para mapear objetos persistentes para um banco de dados. Como exemplo pode-se citar a coleção de padrões para persistência de Yoder *et al.* (1998), que facilita a implementação de sistemas orientados a objetos (OO) com banco de dados relacionais;

•**Padrões de Implementação** ou **Idiomas**: específicos de linguagens de programação, esses padrões descrevem como implementar aspectos particulares dos componentes ou a relação entre eles utilizando as características da linguagem. Como exemplo, pode-se citar o padrão *Counted Point* [Buschmann *et al*., 1996];

•**Padrões de Testes**: descrevem diferentes métodos de testes de sistemas. São exemplos dessa categoria os padrões *Black Box Testing*, *White Box Testing* e *Acceptance Testing*, identificados por Grand (1999).

Entre os autores de padrões de IHC não existe uma definição amplamente aceita. Segundo Borchers (2000), um padrão de projeto captura uma solução comprovada para um problema de projeto recorrente em uma forma de fácil entendimento, gerativa e compreensível às pessoas. Enquanto que Tidwell (1999) define padrões como descrições para possíveis boas soluções para um problema comum de projeto em um certo contexto, descrevendo as qualidades invariáveis de todas as soluções. O termo "qualidades invariáveis" refere-se às características comuns e constantes ao analisar várias aplicações do padrão.

Não se encontra na literatura uma classificação tão clara dos padrões de IHC quanto a encontrada para os padrões de ES [Buschmann *et al*., 1996]. Para a realização deste trabalho foram utilizadas as categorias apresentadas por Alpert (2003):

•**Padrões de Interação Humano-Computador**: relacionados com preocupações de alto nível e algumas vezes com *guidelines*, envolvendo a psicologia do usuário, auxiliando no projeto da interação. Podem ser citados, como exemplos, os padrões da linguagem de padrões *Common Ground* [Tidwell, 1999];

•**Padrões de Interface com o Usuário**: auxiliam no projeto de detalhamento da interface com o usuário, e estão relacionados com problemas de interação específicos. Sua solução é baseada em componentes de interface com o usuário. Como exemplo, pode-se citar os padrões da coleção *UI Patterns & Techniques* [Tidwell, 2003] e os padrões para projeto de *GUI* [Welie, 2003].

As categorias aqui apresentadas foram relacionadas com as etapas do modelo de processo de Prototipação, por meio de um estudo de caso, conforme apresentado na próxima Seção.

3. Aplicando Padrões de ES e de IHC em um Processo de Desenvolvimento

Diversos são os modelos de processo propostos pela ES e pela IHC para o desenvolvimento de sistemas interativos [Sommerville, 2003] [Preece, 1993]. Dentre esses modelos destaca-se o modelo de processo de Prototipação, pois o desenvolvimento de protótipos é parte integral do desenvolvimento de um sistema interativo e por ser uma abordagem altamente participativa [Preece, 1993].

Entretanto, o foco do engenheiro de software difere do foco do especialista em IHC durante a elaboração do protótipo. Enquanto que o engenheiro de software está preocupado em compreender o processo de desenvolvimento, elaborando uma lógica interna para o sistema, o especialista em IHC preocupa-se com os aspectos externos, ou seja, com a interação e a interface com o usuário [Silva *et al*., 2004]. Entretanto é preciso desenvolver o protótipo de tal modo a explorar aspectos de ambas as áreas. Tais fatores influenciaram a escolha do modelo de processo de Prototipação como o primeiro a ser estudado em conjunto com padrões de IHC e de ES.

Após um estudo sobre o modelo de processo Prototipação, uma lista de problemas e considerações foi elaborada, as quais são apresentadas resumidamente na Tabela 1. Ponderando os problemas e as considerações levantadas, em conjunto com as categorias de ES e de IHC descritas anteriormente, elaborou-se o modelo de processo de Prototipação Apoiado por Padrões [Silva *et al*., 2004]. Cada etapa foi relacionada às categorias de padrões que podem ser aplicados para amenizar as considerações levantadas. Ressaltando que os padrões de processo e organizacionais podem ser aplicados em várias etapas, estando, portanto, referenciados do lado externo da Figura.

Planejou-se desenvolver o estudo de caso em duas fases. Durante a primeira fase o sistema foi desenvolvido sem a aplicação de padrões, enquanto que na segunda fase os padrões foram aplicados durante o processo de desenvolvimento, conforme é apresentado na Figura 1. No estudo de caso que é detalhando neste artigo foi desenvolvido um sistema para uma lanchonete, que consiste do gerenciamento dos pedidos realizados pelos clientes através do telefone, e também do controle de itens que podem ser adicionados aos pedidos, tais como ingredientes, taxa de entrega, etc. Deste estudo de caso participaram dois especialistas em desenvolvimento e um usuário final,

Tabela 1 – Considerações levantadas para as etapas do modelo de processo Prototipação

Etapa	Considerações
Coleta e Refinamento de Requisitos	- Procurar por informações que auxiliam a compreender o domínio do problema e os requisitos - Obter os requisitos do sistema - Delimitar os requisitos do sistema por completo, o que é uma das dificuldades do usuário.
Projeto Rápido	- Elaborar um projeto da interface com o usuário que englobe os requisitos a serem refinados, representando os aspectos que são visíveis ao usuário e o nível de conhecimento do usuário. - Considerar as melhores decisões relacionadas à funcionalidade no projeto rápido - Garantir que o processo não será oneroso (tempo e recursos)
Construção do Protótipo	- Minimizar o tempo e os recursos despendidos no desenvolvimento - Assegurar o correto funcionamento do protótipo para que uma avaliação consistente seja realizada na próxima etapa - Englobar somente as funções necessárias, não há necessidade de incluir todas que comporão o produto final.
Avaliação do Protótipo pelo Cliente	- Guiar o usuário para um melhor refinamento dos requisitos desejados, engajando-o e cuidando para que tenha uma comunicação efetiva com o usuário. - Preocupar-se em como obter melhor as informações do usuário - Aproveitar ao máximo o tempo despendido com o usuário na avaliação, a fim de obter informações suficientes para detalhar os requisitos - Fazer um planejamento para permitir a participação dos clientes representativos (usuários finais do sistema), e que o processo de avaliação não seja muito oneroso.
Refinamento do Protótipo	- Coletar críticas e problemas resultantes da avaliação do protótipo, apresentando diretivas de solução para as considerações levantadas
Engenharia do Produto	- Desenvolver o sistema seguindo práticas de ES e de IHC

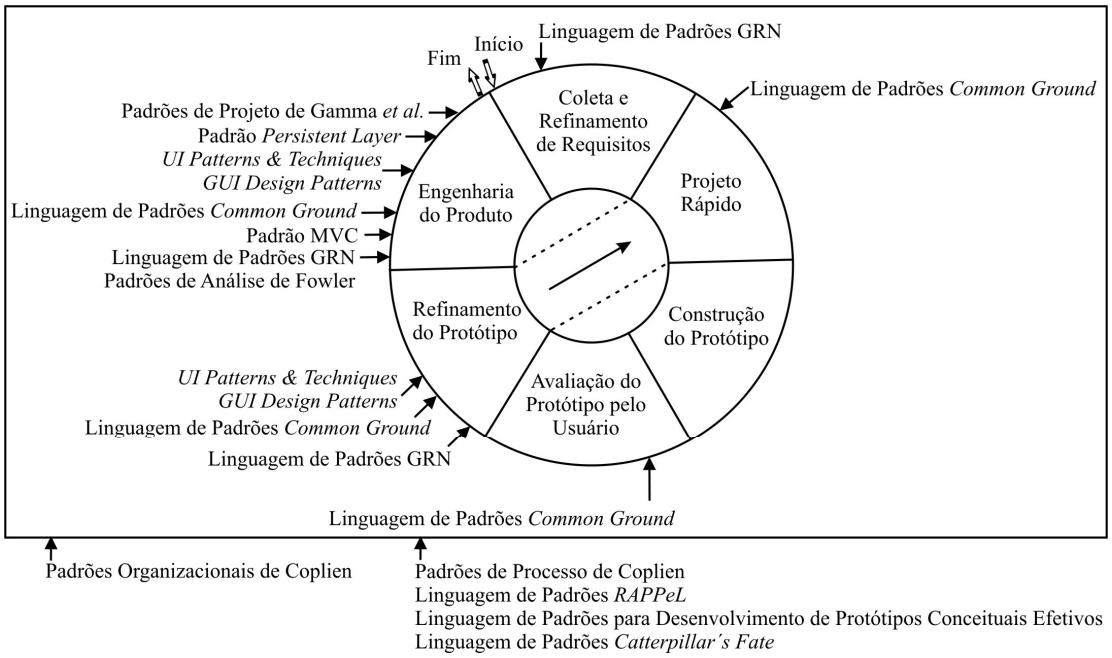


Figura 1 – Instanciação do Modelo de Processo de Prototipação Apoiado por Padrões para desenvolvimento de um sistema para gerência de pedidos de uma lanchonete.

sendo que o especialista que participou da primeira fase não conhecia os padrões, enquanto que o especialista da segunda fase estudou os padrões a serem aplicados.

Diversos padrões de processo que tratam do desenvolvimento de protótipos foram encontrados na literatura. O padrão *Prototype* [Whitenack, 1995] propõe o desenvolvimento de protótipos descartáveis, enquanto que o padrão *Prototypes* [Coplien, 1995] abrange tanto protótipos descartáveis quanto protótipos evolucionários, apresentando as vantagens e as desvantagens de cada um dos tipos de protótipos. Tais padrões comentam sobre o engajamento do cliente durante o uso de protótipos. Portanto, a linguagem de padrões para desenvolvimento de protótipos conceituais efetivos pode ser aplicada em conjunto com tais padrões para engajar o cliente e aumentar a sua participação. Observa-se aqui um relacionamento entre o padrão *Prototypes* e o padrão *Use It and Lose It* [Stimmel, 1999], que tratam sobre o desenvolvimento de protótipos descartáveis. Esses padrões também comentam sobre o uso do protótipo como base para elaborar os casos de uso, o que será discutido posteriormente.

Os padrões que estão relacionados ao engajamento do cliente no desenvolvimento de protótipos são o *Customer Rapport*, *Engage the Client Early* [Stimmel, 1999] e *Come on Baby, Light My Fire*. O padrão *Customer Rapport* apresenta a necessidade de se ter um bom relacionamento com o cliente, focando o usuário e os envolvendo no projeto da interface com o usuário dos protótipos ou do produto final. O padrão *Engage the Client Early* também trata o desenvolvimento de protótipos e a participação do usuário, destacando a diretiva de permitir ao usuário dirigir os esforços do desenvolvimento. Outra consideração é não distrair o usuário com questões particulares da interface com o usuário durante o uso de protótipos, tratado pelo padrão *Come on Baby, Light My Fire*.

Na etapa de Coleta e Refinamento de Requisitos foram aplicados padrões de processo, organizacionais e de análise. Os padrões de processo foram escolhidos seguindo o objetivo dessa etapa (identificar, especificar e validar os requisitos do sistema), entre eles cita-se o padrão *Behavioral Requirements*, que fornece diretivas para a captura de requisitos comportamentais do sistema.

Para a especificação de requisitos foi aplicado o padrão *Requirements Specification*, que apresenta diretivas para a elaboração de um documento de especificação de requisitos, comentando sobre a adoção de um modelo de especificação que contemple tanto os requisitos identificados quanto os artefatos elaborados, e a validação do documento de requisitos com o cliente. Entretanto, este padrão não define quando finalizar o desenvolvimento do protótipo e iniciar a elaboração do documento de especificação de requisitos. Para tal consideração, pode-se aplicar o padrão *Let's Make a Deal*, que apresenta diretivas de quando iniciar a elaboração de tal artefato.

Para a validação foi aplicado o padrão *Requirements Validation,* que apresenta informações a serem consideradas para a realização de reuniões de validação dos requisitos. Porém, esse padrão não apresenta uma abordagem, ou passos, que contemple todo o processo de validação, desde o planejamento das reuniões até a efetivação das mudanças solicitadas. Para amenizar essa consideração, o padrão *Technical Review* [Ambler, 1998] pode ser aplicado após a aplicação do padrão *Requirements Validation*.

Durante a realização do estudo de caso sem aplicar padrões, percebeu-se que somente o propósito do sistema é insuficiente para elaborar questões para o levantamento de requisitos. São necessárias mais informações para se evitar um numero excessivo de retorno ao usuário. Assim, parte dos padrões da linguagem de padrões GRN [Braga *et al*., 1999] foi aplicada como guia para se elaborar questões ao usuário. Para escolher os padrões foram utilizados como critério o propósito do sistema, seguindo os relacionamentos apresentados na linguagem de padrões.

Após os requisitos identificados serem descritos no documento de especificação, conforme o padrão *Requirements Specification*, deu-se início a etapa de Projeto Rápido, na qual, foram elaborados *mockups* das interfaces com o usuário. Nessa etapa foram aplicados os padrões *Prototypes* e *User Interface Requirements,* objetivando a elaboração de um protótipo para auxiliar no levantamento e refinamento dos requisitos.

Segundo o padrão *Prototypes,* inicialmente deve-se elaborar um protótipo de baixa fidelidade com a participação do cliente, e em seguida, elaborar protótipos de alta fidelidade, caso seja necessário. Para a elaboração do protótipo de baixa fidelidade foi aplicada a abordagem de Dearden *et al.* (2002), que aplica padrões de IHC para melhorar a comunicação entre os usuários e os especialistas. Protótipos de alta fidelidade foram elaborados e avaliados em etapas posteriores. A abordagem de Dearden *et al.* (2002) é dividida em três passos **(I) Introdução**: o facilitador introduz os conceitos de padrões e linguagem de padrões ao usuário final; e **(II) Leitura dos Padrões**: o facilitador solicita ao usuário final a ler os padrões, retirando dúvidas do usuário final; (**III) Desenvolvimento da Interface**: por meio de prototipação em papel, o usuário final, em conjunto com o facilitador, elabora as interfaces do sistema utilizando padrões para se expressar. Após elaborar uma tela, o facilitador verifica se essa satisfaz todas as diretivas dos padrões aplicados.

Um subconjunto de 29 padrões da linguagem de padrões *Common Ground* foi apresentado ao usuário durante a elaboração do projeto da interface com o usuário, permitindo que ele expressasse seus anseios. A linguagem de padrões não foi apresentada em sua completitude ao usuário devido à necessidade de aprendizado e de tradução dos padrões, pois o usuário não conhece a língua inglesa. Dessa forma, somente os padrões diretamente relacionados ao estudo de caso foram utilizados.

Na Figura 2 é apresentada a tela de cadastro de clientes e pedidos que foi projetada pelo usuário. Os padrões de interação humano-computador aplicados para a elaboração dessa tela também são apresentados nessa Figura. Para o projeto dessa tela foi aplicado, primeiramente, o padrão *High-Density Information Display* (1). Esse padrão é um dos três que podem ser aplicados para definir a forma básica do conteúdo a ser apresentado, que é a primeira consideração no projeto da interface com o usuário ao utilizar a linguagem de padrões *Common Ground* [Tidwell, 1999]. Para a escolha de qual dos três padrões aplicar foram considerados os periféricos de saída existentes, visto que o monitor pode exibir uma vasta quantidade de informações em diferentes resoluções e tamanhos, e o fato de o usuário interagir constantemente com o sistema.

Em seguida, foram aplicados os padrões *Navigable Spaces* (2), pois o sistema é composto por diversas telas e o usuário necessita navegar entre elas, e *Tabular Set* (3), que define a apresentação de informações através de uma tabela (aplicado, nesse exemplo, para apresentar os pedidos solicitados). Seguindo os relacionamentos da

linguagem, em seguida foi aplicado o padrão *Go Back to a Safe Place* (4), resultando no botão fechar na parte superior direita da janela. Também foi aplicado o padrão *Pointer Shows Affordance* (5), resultando na mudança do ponteiro do *mouse* para *I-Bean* (I) quando o ponteiro está sobre um campo de texto editável.

Para o detalhamento da interface com o usuário foi aplicado o padrão *Form* (6). Esse padrão define diretivas para a apresentação de um formulário, na qual o usuário poderá entrar com os dados necessários para realizar a tarefa. Nesse exemplo, foram definidos caixas de textos para a entrada de dados sobre o cliente. Esse padrão também foi aplicado, em conjunto com o padrão *Tabular Set* (3), para a entrada de dados relacionados aos pedidos.

Após a aplicação do padrão *Form* (6), foram aplicados os padrões *Forgiving Text Entry* (7), *Structured Text Entry* (8), *Choice from a Large Set* (9) e *Small Group of Related Things* (10). O Padrão *Forgiving Text Entry* e *Structured Text Entry* foram aplicados para alguns campos de texto onde existe certa estrutura do dados como, por exemplo, para o campo "Telefone". Nesse caso, o usuário não é obrigado a digitar os caracteres fixos da estrutura (padrão *Forgiving Text Entry*) e quando não existe dado fornecido pelo usuário, o campo apresenta a formatação do telefone esperada (padrão *Structured Text Entry*). O padrão *Choice from a Large Set* foi aplicado para o campo "Estado", na qual o número de opções é maior que dez e o usuário necessita escolher um valor específico. O padrão *Small Group of Related Thing* foi aplicado, novamente, para organizar a localização espacial dos campos para a entrada dos dados do cliente.

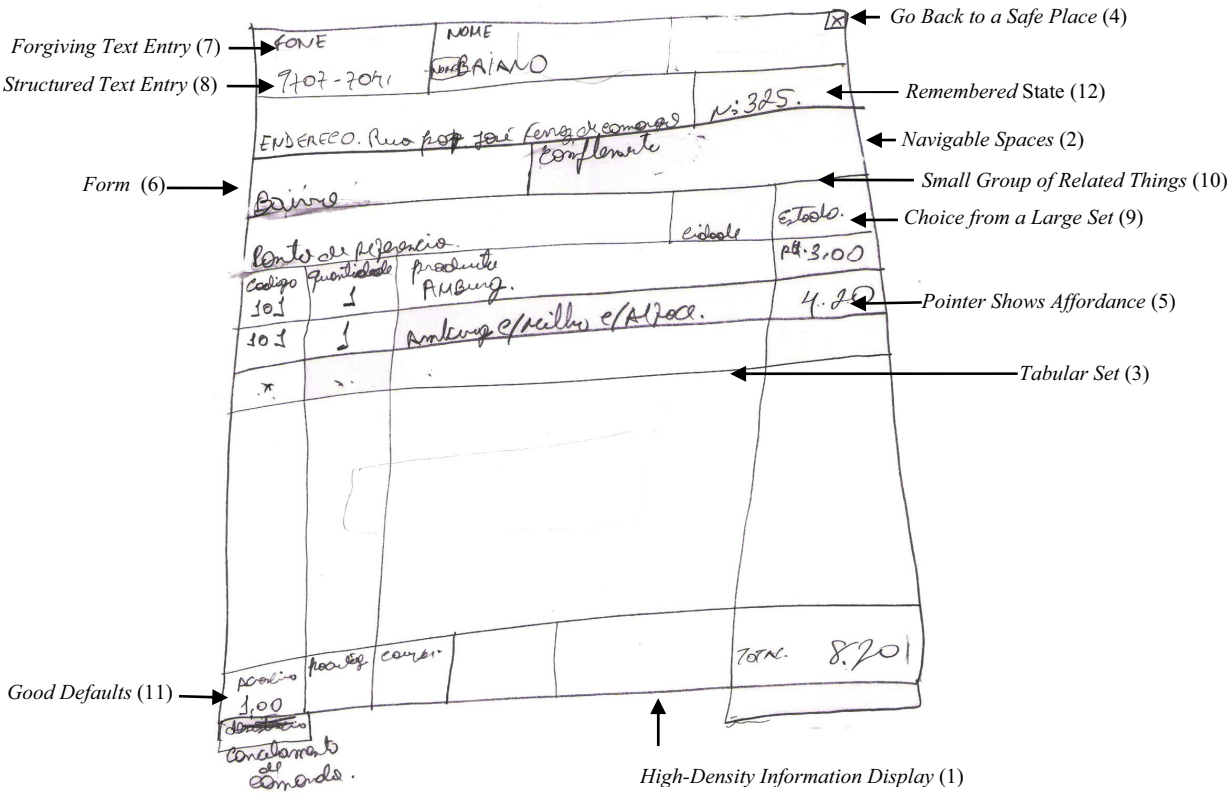


Figura 2 – Padrões aplicados para desenvolver a tela de cadastro de clientes e pedidos.

Aplicando o padrão *Good Defaults* (11), valores foram definidos para alguns campos do formulário como, por exemplo, o campo "Acréscimo", que considera a taxa de entrega. O padrão *Remembered State* (12) foi aplicado para garantir que os valores dos campos do formulário não sejam alterados, caso o usuário navegue para outra tela ou para outro programa.

Devido à dinâmica da técnica de Prototipação em Papel, o usuário manifestou seus interesses no projeto da interface com o usuário, empregando parte dos conhecimentos que absorveu do aprendizado de padrões. As interfaces elaboradas foram avaliadas pelos especialistas verificando se atendiam ou não as diretivas de cada um dos padrões. Para as diretivas que não foram atendidas, perguntas foram elaboradas para que as telas projetadas respeitassem todas as diretivas dos padrões selecionados. Esta avaliação possibilitou a diminuição do número de decisões de projeto que estavam sendo realizadas na etapa de Construção do Protótipo, devido ao esquecimento de detalhes sobre a interface.

Observou-se, deste modo, que os padrões de IHC complementam a aplicação dos padrões *Prototypes* e *User Interface Requirements*, fornecendo diretivas para elaborar a interação e a interface com o usuário e um vocabulário de comunicação entre o usuário e os especialistas.

Na seqüência, um protótipo foi implementado usando a linguagem de programação *Visual Basic*, e foi avaliado pelo usuário durante a etapa de Avaliação do Protótipo pelo Usuário. Para essa etapa houve uma atividade de planejamento, com o objetivo de definir as tarefas a serem realizadas pelo usuário durante a avaliação. Durante a execução da avaliação, o protótipo era manipulado pelo usuário, sendo observado pelos especialistas durante a realização das tarefas elaboradas. Observou-se, durante a interação, que o usuário notou a aplicação dos padrões de IHC no protótipo. Algumas vezes, ao descrever um problema encontrado no protótipo, ele tentava fornecer uma solução através dos padrões aprendidos anteriormente.

Após a etapa de Avaliação do Protótipo pelo Usuário, durante a etapa de Refinamento do Protótipo, uma lista foi elaborada com as críticas apresentadas pelos usuários e pelos especialistas. Novamente os especialistas utilizaram padrões para corrigir essas deficiências. Por exemplo, na interface apresentada anteriormente (Figura 2) o usuário não citou a necessidade de confirmação de algumas operações. Mas no caso de cancelamento de comandas notou a necessidade da confirmação do cancelamento de pedidos para a prevenção de erros. Nesse caso o padrão *Shield* [Welie, 2003] pode ser aplicado.

No planejamento do estudo de caso foram definidas duas iterações no uso do modelo de processo Prototipação antes da realização da etapa de Engenharia do Produto, objetivando identificar novos requisitos e refinar os requisitos já identificados. A aplicação de padrões durante a segunda iteração foi semelhante à primeira, com exceção da etapa de Projeto Rápido. Nessa etapa não foi elaborado um novo projeto para a interface com o usuário, realizando-se somente modificações no projeto existente, conforme as necessidades levantadas na etapa de Refinamento do Protótipo.

A primeira atividade realizada na etapa de Engenharia do Produto é a análise, na qual foram elaborados os modelos de casos de uso, de classe e de seqüência. Para a realização dessa atividade foram aplicados os padrões de processo *Behavioral*

Requirements, que apresentam diretivas para a descrição dos comportamentos do sistema através de casos de uso, em conjunto com o padrão *Scenarios Define Problem,* que aconselha a elaboração de casos de uso para documentar os comportamentos e comunicar-se com o cliente. Em seguida, foram aplicados alguns padrões para escrita de casos de uso efetivos [Adolph *et al*., 2002], procurando complementar as diretivas fornecidas pelos padrões de processo aplicados. Essa aplicação conjunta resultou na identificação de outros três relacionamentos entre os padrões de ES, que serão discutidos na seção seguinte.

Na fase de Análise também foi aplicado o padrão de processo *Problem Domain Analysis*, que define um conjunto de perguntas a serem respondidas para realizar a análise. Em seguida, para auxiliar a identificar os objetos do domínio, os relacionamentos entre os objetos, seus atributos e seus métodos, foram aplicados os padrões a linguagem de padrões GRN [Braga *et al*., 1999] em conjunto com o padrão *Party* [Fowler, 1996]. Observa-se aqui um relacionamento entre o padrão de processo *Problem Domain Analysis* e padrões de análise, os quais podem ser aplicados para auxiliar a responder as questões desse padrão de processo.

Durante a etapa de Projeto, para definir os passos da elaboração dos objetos responsáveis pela interface com o usuário, foi aplicado o padrão *Human Interface Role Is a Special Interface Role*. Esse padrão de processo fornece como diretiva a separação dos objetos da interface com o usuário dos demais objetos, aplicando, por exemplo, o padrão MVC. Esse padrão também indica a elaboração de diversas alternativas de projeto para serem discutidas com a equipe, selecionando em seguida as melhores alternativas. Podem ser aplicados padrões de IHC para auxiliar a definir boas alternativas. Observou-se um outro relacionamento entre o padrão *Human-Interface Role Is a Special Interface Role* e os padrões de IHC, que será discutido na seção seguinte.

Como padrão arquitetural foi adotado o padrão MVC. Durante sua aplicação foi empregado também o padrão *Observer*, que complementa a solução do padrão MVC, e os padrões de IHC. Buschmann *et al*. (1996) apresentam dez passos para a implementação do padrão MVC, sendo que o terceiro é o projeto da parte da visualização da interface com o usuário, representado pela *View*. Para auxiliar a realizar esse passo podem ser aplicados padrões de IHC. Nesse estudo de caso, os mesmos padrões de IHC aplicados no protótipo foram aplicados no produto final, pois a Prototipação se mostrou útil como um mecanismo para averiguar quais padrões de IHC deveriam estar presentes no produto final.

Relacionamentos entre os padrões de IHC e padrões de projeto foram identificados durante a atividade de projeto. Por exemplo, cita-se o requisito de interface com o usuário no qual o botão “Cancelar Comanda” (Figura 2) só deve está habilitado se houver algum pedido sendo realizado, resultante da aplicação do padrão *Disabled Irrelevant Things*. O botão é inicialmente desabilitado, pois não existem pedidos sendo realizados. Após o usuário fornecer o telefone do cliente, o sistema permite a edição do pedido e, então, habilita o botão para cancelar os pedidos solicitados. Percebe-se que um objeto que pode ser desabilitado ou habilitado está relacionado a um contexto composto por um ou mais objetos. O padrão *Observer* pode ser aplicado para que, quando o contexto de tal objeto alterar, o objeto seja informado sobre tal mudança, verificando, assim, se ele é relevante ou não para o novo contexto e

alterando sua permissão de manipulação por parte do usuário. Conclui-se que o padrão de projeto *Observer* auxilia a projetar o padrão de interação humano-computador *Disabled Irrelevant Things*.

Por fim, o produto final foi implementado na linguagem Java, respeitando o projeto da interface com o usuário definido durante o desenvolvimento e avaliação do protótipo.

4. Relacionando Padrões de ES e de IHC

Durante a realização do estudo de caso percebeu-se que padrões de ES e de IHC se complementam durante o desenvolvimento de um sistema interativo. Vários relacionamentos foram identificados por meio da leitura dos padrões e validados com o estudo de caso descrito na seção anterior. Na Tabela 2 são apresentados o relacionamento dos padrões de IHC que complementam os padrões de ES. Tais relacionamentos foram identificados durante a realização dos estudos de caso.

Observando os relacionamentos identificados entre os padrões de IHC complementando os padrões de ES, percebe-se que os relacionamentos partem dos padrões de processo de ES que tratam de alguma forma o desenvolvimento de interfaces com o usuário, seja durante a elaboração de protótipos (padrão Prototypes [Coplien, 1995]), seja na elaboração do produto final (padrão MVC [Buschmann et al., 1996], aplicado durante a definição da arquitetura do sistema, e o padrão User Interface Role Is a Special Interface Role [Kerth, 1995], aplicado durante a definição da solução para o sistema, especificando através de um projeto de software).

Tabela 2 – Relacionamento dos padrões de ES complementados pelos padrões de IHC

Padrão de ES	Relacionamento
Prototypes	Padrões de IHC complementam o padrão *Prototypes*, pois fornecem diretivas para elaboração da interação e do *layout* de sistemas interativos, inclusive para os protótipos, que podem ser desenvolvidos resultantes da aplicação do padrão *Prototypes*.
Human Interface Role is a Special Interface Role	Padrões de IHC podem ser aplicados em conjunto com esse padrão para auxiliar a definir uma boa solução ao se realizar o projeto da interface com o usuário, resultante da aplicação desse padrão de ES. Os relacionamentos entre padrões de IHC com os padrões de projeto podem auxiliar a definir responsabilidades dos objetos da interface com o usuário que, segundo esse padrão de ES, devem ser identificados.
MVC	No projeto das Visões (*View*), que representa a interface com o usuário, é possível aplicar padrões de IHC.

Empregando os estudos de caso, também foram coletados relacionamentos entre os padrões de ES que complementavam os padrões de IHC. A Tabela 3 sumariza os relacionamentos identificados.

Na Tabela 4, é apresentada uma síntese dos relacionamentos identificados entre os padrões de ES, identificados através dos estudos de caso realizados. Diversos outros relacionamentos entre padrões de ES foram identificados e apresentados por seus autores, principalmente os relacionamentos com os padrões de projeto, conforme é possível perceber durante a leitura de tais padrões.

Tabela 3 – Relacionamento dos padrões de IHC complementados pelos padrões de ES

Padrão de IHC	Padrão de ES	Relacionamento
Composed Command	*Little Language*	O padrão *Composed Command* apresenta diretivas para a linguagem que será utilizada pelo usuário para a interação com o sistema. O padrão *Little Language* define como objetos colaboram para analisar e realizar a ação correspondente ao comando fornecido.
Undo	*Memento* + *Command*	O padrão *Undo* apresenta diretivas para disponibilizar a operação de desfazer, enquanto que o padrão *Command* define uma interface para os comandos possíveis (e com isso é possível elaborar uma lista de operações a desfazer) e o padrão *Memento* realiza a operação desfazer em si, retornando o objeto ao estado anterior.
Tabular Set	*Iterator*	O padrão *Tabular Set* apresenta diretivas para apresentar os dados por meio de uma tabela. Entretanto, é desejável que o objeto que apresenta a tabela não dependa do modo como os dados sejam representados. A aplicação do padrão *Iterator* permite essa independência.
Step-by-Step Instructions	*Memento*	Uma das diretivas do padrão *Step-by-Step Instructions* é fornecer a possibilidade do usuário retornar a um passo. Possivelmente um passo realizado altera o estado de um ou mais objetos. O padrão *Memento* permite que o estado anterior do objeto seja recuperado ao retornar um passo sem que o encapsulamento seja violado.
Disabled Irrelevant Things	*Observer*	Um determinado objeto que pode se tornar irrelevante está relacionado ao contexto que define se ele é irrelevante ou não. Esse contexto pode ser formado por outros objetos. O padrão *Observer* pode ser aplicado para informar ao objeto que houve mudanças em seu contexto. Quando o objeto é informado sobre a mudança do seu contexto, ele é capaz de determinar se é irrelevante ou não nesse novo contexto.

Tabela 4 – Relacionamento dos padrões de ES complementados pelos padrões de ES

Padrão de ES	Padrão de ES	Relacionamento
Customer Rapport	*Engage the Client Early*	O padrão *Customer Rapport* apresenta diretivas para estabelecer um bom relacionamento com o cliente, focando os usuários e envolvendo-os no projeto da interface com o usuário em conjunto com a elaboração de protótipos. O padrão *Engage the Client Early* também trata do desenvolvimento de diversos protótipos, considerando engajar o usuário e sua participação, permitindo-o guiar os esforços do desenvolvimento.
Scenarios Define Problem	*Behavioral Requirements*	O padrão *Scenarios Define Problem* propõe, como solução para o problema dos documentos de projeto serem veículos ineficientes para comunicação com o usuário, o emprego de casos de uso. O padrão *Behavioral Requirements* aprofunda essa questão e apresenta também algumas diretivas para a elaboração dos casos de uso.
Customer Rapport	*Come on Baby, Light My Fire*	O padrão *Customer Rapport* apresenta diretivas para estabelecer um bom relacionamento com o cliente, focando os usuários e envolvendo-os no projeto da interface com o usuário, em conjunto com a elaboração de protótipos. O padrão *Come on Baby, Light My Fire* também trata do desenvolvimento de protótipos, considerando engajar o usuário e sua participação, mas sem distraí-lo com questões particulares de interface.
Prototypes	*Prototype*	O padrão *Prototypes* apresenta diretivas para elaborar protótipos descartáveis ou evolucionários. O padrão *Prototype* comenta sobre a elaboração de protótipos descartáveis.

Tabela 4 – Relacionamento dos padrões de ES complementados pelos padrões de ES (continuação)

Padrão de ES	Padrão de ES	Relacionamento
Prototypes	*Use It and Lose It*	O padrão *Prototypes* apresenta diretivas para elaborar protótipos descartáveis ou evolucionários. Para o desenvolvimento de protótipos descartáveis pode-se aplicar em seguida o padrão *Use It and Lose It* que também fornece diretivas para a elaboração de protótipos descartáveis, considerando o desenvolvimento rápido do protótipo e o engajamento do cliente.
Prototype	*Use It and Lose It*	O padrão *Prototype* comenta sobre a elaboração de protótipos descartáveis para auxiliar a compreender os requisitos. Diversos fatores estão relacionados ao desenvolvimento de protótipos, inclusive a redução do tempo de desenvolvimento do protótipo, como apresenta o padrão *Use It and Lose It*.
Let's Make a Deal	*Requirements Specification*	O padrão *Let's Make a Deal* apresenta diretivas informando quando a elaboração de protótipos pode ser finalizada e a elaboração de um documento de requisitos apropriado pode ser iniciada. O padrão *Requirements Specification* define diretivas para a elaboração de um documento de requisitos.
Requirements Validation	*Technical Review*	O padrão *Requirements Validation* comenta que todos os interessados devem ler o documento de requisitos em reuniões de revisão. O padrão *Technical Review* apresenta diretivas para o planejamento, execução e coleta de resultados de reuniões para revisão de um artefato.
Problem Domain Analysis	Padrões de Análise	O padrão *Problem Domain Analysis* comenta sobre a necessidade de elaborar uma representação para o domínio do sistema, levando em conta questões comuns que são deparadas durante a realização de uma análise. Para apoiar a realização da análise podem ser aplicados padrões de análise encontrados na literatura.
Behavioral Requirements	Padrões de Caso de Uso	O padrão *Behavioral Requirements* apresenta diretivas para representar o comportamento do sistema por meio de casos de uso. Ele também apresenta algumas diretivas para a elaboração destes casos de uso. Caso sejam necessárias mais diretivas, é possível aplicar os padrões voltados para a elaboração de casos de uso.
Scenarios Define Problem	Padrões de Caso de Uso	O padrão *Scenarios Define Problem* propõe a elaboração de casos de uso, como solução para o problema dos documentos de projeto, ser de difícil compreensão pelo usuário. Para escrever casos de uso que permitam uma leitura fácil ao usuário, entre outras características, pode-se aplicar os padrões para elaboração de casos de uso.

5. Conclusões e Trabalhos Futuros

Como resultado deste trabalho, que está baseado em um projeto que visa integrar as visões de IHC e de ES no desenvolvimento de sistemas interativos, apresentamos 19 relacionamentos coletados durante a aplicação de um conjunto de padrões de ES e de IHC em três estudos de caso. Acredita-se que um maior número de relacionamentos podem ser identificados através da aplicação dos padrões e da análise destas aplicações no desenvolvimento de sistemas interativos.

Realizar a identificação de relacionamentos entre os padrões das duas áreas é uma tarefa árdua, pois existem muito padrões à considerar. Entretanto, acredita-se que esses esforços trazem benefícios, pois por meio da linguagem proposta é possível um melhor aproveitamento dos benefícios que os padrões trazem ao serem aplicados em um processo de desenvolvimento, realizando a transferência de conhecimento entre os participantes de níveis diferentes e facilitando a comunicação entre eles. Relacionar os padrões das duas áreas também é útil para motivar os especialistas de ambas as áreas a

desenvolver o sistema em parceria, o que muitas vezes não ocorre devido à falta de comunicação, divergência de foco e por possuírem formação diferentes.

Os seguintes resultados foram alcançados: (1) validação da proposta do modelo de processo Prototipação Apoiado por Padrões; (2) identificação de 19 relacionamentos entre padrões de ES e de IHC; (3) validação da proposta da existência de relacionamentos entre padrões de IHC e de ES. Através da avaliação desses resultados, concluímos que: (1) padrões de ES e de IHC podem se complementar para desenvolver sistemas interativos de forma mais abrangente, tratando aspectos de ambas as áreas; (2) relevância na identificação de relacionamentos entre padrões de ES e de IHC.

Como trabalhos futuros pretende-se elaborar uma linguagem para desenvolvimento de sistemas interativos que considere padrões de ambas as áreas a partir da coleta dos relacionamentos entre os padrões, incluindo os aqui apresentados. Entretanto, percebe-se que, para chegar a uma linguagem de padrões que considere tal quantidade de padrões, é necessário que a linguagem [Meszaros e Double 1996]: 1) apóie todos os aspectos importantes em um dado domínio, 2) forneça uma tabela resumindo os padrões passíveis de serem empregados (padrão *Problem/Solution Summary*), 3) utilize um mesmo exemplo em toda a linguagem (padrão *Running Example*), 4) ofereça um glossário de termos (padrão *Glossary*), e 5) descreva os relacionamentos dentro do texto que descreve o padrão (padrão *Pattern Language*). Todas essas questões serão consideradas em trabalhos futuros.

Referências

Adolph, S., Bramble, P., Cockburn, A., Pols, A. Patterns for Effective Use Cases, Pearson Education, Inc., EUA, 2002.

Alpert, S. R. (2003) “Getting Organized: Some Outstanding Questions and Issues Regarding Interaction Design Patterns”, In: Workshop on “Perspectives on HCI Patterns” at CHI, 20., 2003.

Ambler, S., Process Patterns: Building Large-Scale Systems Using Object Technology, Cambridge University Press, 1998.

Borchers, J. O. (2000) “CHI Meets PLoP: An Interaction Patterns Workshop”. In: SIGCHI Bulletin, Nova Iorque, EUA, v. 32, n. 1, p. 9-12.

Braga, R. T., Germano, F. S. R. and Masiero, P. C. (1999) “A Pattern Language for Business Resource Management”, In: Pattern Languages of Programming Conference, 6., Monticello, EUA.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., Pattern-Oriented Software Architecture Volume 1: A System of Patterns, John Wiley & Sons Ltd., 1996.

Coplien, J. O. (1995) “A Generative Development-Process Pattern Language”, In: Pattern Language of Programming Design, Edited by J. O. Coplien and D. C. Schmidt, EUA, Addison Wesley Longman Inc.

Dearden, A., Finlay, J., Allgar, E. and McManus, B. (2002) “Using Pattern Languages in Participatory Design”. Proceedings of Participatory Design Conference, (2002).

Fowler, M., Analysis Patterns: Reusable Object Models, Addison Wesley, 1996.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

Grand, M., Patterns in Java Volume 1, John Wiley & Sons Inc., 1998.

Grand, M., Patterns in Java Volume 2, John Wiley & Sons Inc., 1999.

Kerth, N. L. (1995) "Caterpillar´s Fate: A Pattern Language for the Transformation from Analysis to Design", In: Pattern Language of Programming Design, Edited by J. O. Coplien and D. C. Schmidt, EUA, Addison Wesley Longman Inc.

Meszaros, G., and Doble, J. (1996) "Metapatterns: A pattern language for pattern writing", Proceedings of the 3rd Pattern Languages of Programming Conference, 1996.

Preece, J., A Guide to Usability: Human factors in computing. Addison-Wesley Pub. Co., Reading, MA, EUA, 1993.

Silva, A. C. da, Silva, J. C. A., Penteado, R. A. D. and Silva, S. R. P. da (2004) "Integrando a Visão da ES e da IHC através da Aplicação de Padrões sobre o Modelo de Prototipação", In: Simpósio Brasileiro de Fatores Humanos em Sistemas Computacionais, 6., Curitiba-PR, Brasil.

Sommerville, I. Engenharia de Software, 6.ed. Addison-Wesley Pub. Co., São Paulo, SP, Brasil, 2003.

Stimmel, C. L. (1999) "Hold Me, Thrill Me, Kiss Me, Kill Me Pattern Language for Developing Effective Concept Prototypes", In: Pattern Languages of Programming Conference, 6., Monticello, EUA.

Tidwell, J. (1999) "Commond Ground: a Pattern Language for Human-Computer Interface Design", http://www.mit.edu/~jtidwell/interaction_patterns.html.

Tidwell, J. (2003) "User Interface Patterns and Techniques", http://time-tripper.com/uipatterns.

Welie, M. van, (2003) "Pattern in Interaction Design", http://www.welie.com.

Whitenack, B. (1995) "RAPPeL: A Requirements-Analysis-Process Pattern Language for Object-Oriented Development", In: Pattern Language of Programming Design, Edited by J. O. Coplien and D. C. Schmidt, EUA, Addison Wesley Longman Inc.

Yoder, J. W., Johnson, R. E. and Wilson, Q. D., (1998) "Connecting Business Objects to Relational Database", In: Pattern Languages of Progamming Conference, 5., Monticello, USA.

Aplicando Padrões de Gerência de Configuração de Software em Projetos Geograficamente Distribuídos[1]

Dario André Louzado, Lucas Carvalho Cordeiro

Siemens Com Mobile Devices – Siemens Eletroeletrônica S.A.
Manaus – AM – Brazil

{dario.louzado,lucas.cordeiro}@siemens.com

Abstract: *Software Configuration Management (SCM) plays an important role in software development projects by controlling the consistency of artifacts during the whole project life cycle. In this article we discuss how a well-known SCM pattern language was applied in a medium size outsourced project. For each applied pattern we explain the context, problem, solution and resulting context. We present the ideas by looking to the SCM system as continuous evolving system as the patterns are applied. Special nuances, common to the complex world of outsourcing, are also emphasized.*

Keywords: *patterns, software configuration management, outsourcing.*

Resumo: *Gerência de Configuração de Software (SCM) desempenha um papel importante no desenvolvimento de projetos de software, controlando a consistência dos artefatos ao longo do ciclo de vida do projeto. Neste artigo é discutido como uma linguagem de padrões de SCM conhecida foi aplicada em um projeto terceirizado de tamanho médio. Para cada padrão aplicado são apresentados contexto de aplicação, problemas enfrentados, soluções consideradas assim como o contexto resultante. As idéias são apresentadas olhando para o sistema de SCM como um sistema que evolui na medida em que os padrões são aplicados. Nuances especiais, comuns ao complexo mundo da terceirização, são também enfatizadas.*

Palavras-chave: *padrões, gerência de configuração de software, terceirização.*

1. Introdução

Uma disciplina da engenharia de software que vem ganhando crescente destaque em projetos de software é a *gerência de configuração do software*, ou *software configuration management* – SCM. A razão para tanto destaque é muito simples. Se entendermos todo o processo de desenvolvimento de software como um software [1], SCM pode ser vista como o subsistema de entrada-saída (*I/O*) deste software.

Indo um pouco mais além, SCM responde pelo controle transacional dos artefatos de software, isto é, pelo controle da consistência do produto de trabalho produzido pelos desenvolvedores ao longo de todo o ciclo de vida do projeto. Por exemplo, ao receber o código-fonte de diferentes desenvolvedores, o gerente de configuração do software organiza este material em um espaço de trabalho (*workspace*) checa as consistências e dispara o processo de geração de *builds*. Como resultado deste último processo, tem-se uma versão *intermediária,* ou incremento, do software em construção ou manutenção.

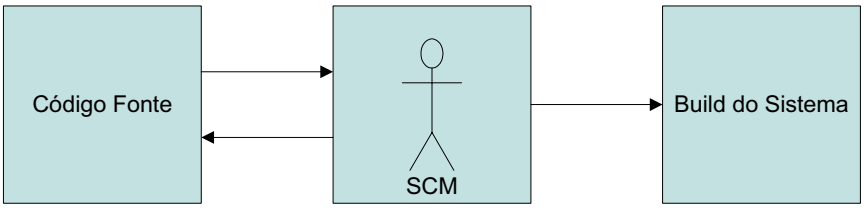


Figura 1. Sistema de entrada e saída SCM

Conforme mostrado na figura 1, o sistema de *I/O* SCM recebe código-fonte (entrada), organiza, verifica as diferenças e consistências e produz um *build*, uma saída, portanto, deste processo. O exemplo é simples, mas serve para ilustrar o papel da gerência de configuração em projetos de software. Um *build* que apresenta problemas de compilação ou integração poderia sacrificar um ou mais dias de trabalho de uma equipe inteira de desenvolvedores ou testatores. *Builds* com este tipo de problema dificultam inclusive a gerência do projeto pelos líderes, pois a noção de progresso é consideravelmente ofuscada.

Na prática, as atividades, considerações e verificações realizadas pelo gerente da configuração são bem menos triviais que o exemplo acima. Neste artigo, estaremos navegando pela linguagem de padrões de gerência de configuração definida em [3], objetivando discutir, para cada padrão aplicado e para a linguagem como um todo, quais as considerações, dificuldades e soluções realizadas no contexto de um projeto real vivenciado pelos autores.

É importante salientar que o propósito principal deste artigo é focar em assuntos estratégicos de SCM tais como: *práticas, políticas e organização.* Com este objetivo em mente, o apêndice A fornece uma visão geral das ferramentas utilizadas no projeto. Além disso, por questões de confidencialidade, o artigo abordará apenas do uso da técnica, omitindo qualquer tipo de informação que envolva o escopo e a estratégia do projeto analisado.

2. Contexto Geral

O projeto analisado tem por objetivo produzir um software de uso *desktop* (com aproximadamente 100,000 LOC) destinado a usuários finais de aparelhos celulares. Por ser destinado ao usuário final em um mercado de massa, trata-se de um projeto crítico o qual demanda um controle rigoroso na configuração do software. A Siemens Communications é uma organização com uma evidente política de presença no mercado (*time-to-market),* o que evidencia a necessidade de controle sobre o projeto, status da configuração e da qualidade do código.

Adicionalmente, o projeto é desenvolvido por quatro parceiros situados fisicamente em localidades diferentes – Figura 2. Esta realidade demanda uma boa comunicação e uma definição clara de responsabilidades, situando ainda mais a gerência da configuração do software como um instrumento chave neste processo.

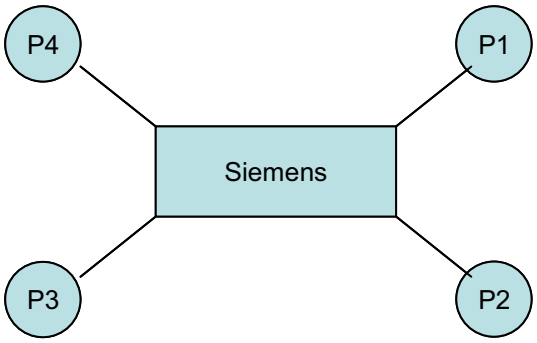


Figura 2. Siemens e seus parceiros

Outra particularidade encontrada foi a necessidade de separar um pouco a gestão dos artefatos de software (atividades de integração e geração do *build*). A primeira é atribuição do gerente de configuração (C*onfiguration Manager*, doravante denominado CM). A segunda é atribuição do gerente de build (*Build Manager*, doravante denominado BM).

3. Organização, arquitetura e gerência de configuração de software

Organizações estruturam-se de acordo com o mercado para lançar produtos ou soluções [2]. Esta estruturação influência fortemente a arquitetura do software. Conforme os sistemas de software tornam-se mais complexos, a arquitetura passa a influenciar a organização e suas decisões.

Uma influência importante é a *localização* do trabalho, isto é, como um pacote de trabalho é atribuído a uma determinada equipe. Dois componentes com muita proximidade e dependência são atribuídos de forma localizada a um time de desenvolvedores, minimizando a demanda por canais de comunicação. Esta abordagem implica em mais agilidade e melhor gestão dos riscos ao projeto.

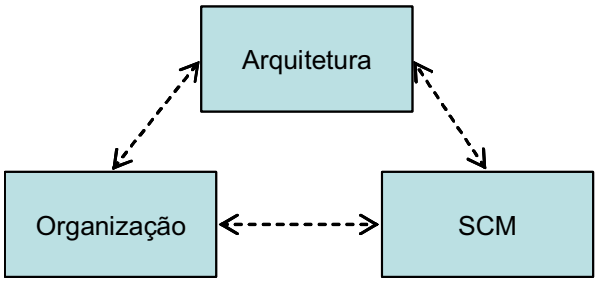


Figura 3. Influências entre organização, arquitetura e SCM

De uma maneira geral, o padrão *Architecture Follows Organization* [2] discute como a arquitetura estrutura os canais de comunicação em uma organização. Continuando com o cliclo de influências, conforme figura 3, tanto as estruturas organizacional quanto arquitetural influenciam diretamente as práticas, políticas, planejamento e as ferramentas destinadas à gerência da configuração do software. Esta fornece, portanto, uma base de sustentação para as outras.

Estudando estas dependências conceituais, foi desenvolvida uma linguagem de padrões destinada à gerência da configuração [3]. Esta linguagem classifica os padrões em duas categorias:

- *Codeline:* padrões relacionados ao controle de versão e ao isolamento de iniciativas distintas de desenvolvimento em linhas de codificação isoladas, tipicamente implementadas em ferramentas de controle de versão com o conceito de *branches*.
- *Workspace:* padrões relacionados ao agrupamento de versões específicas de artefatos do projeto em áreas de trabalho a fim de suportar diferentes atividades do projeto. Exemplos: gerar um *build*, executar *smoke tests*, desenvolver funcionalidades, integrar software, coletar métricas de código, entre outras.

A linguagem de padrões SCM vem sendo refinada há pelo menos cinco anos pelos autores originais e este artigo adiciona uma contribuição a partir de uma experiência em projeto com times geograficamente distribuídos. A figura 4 mostra as interações entre os padrões da linguagem em estudo.

É importante observar a partir do mapa de linguagem de padrões SCM (figura 4), que a seta padrão A ⟶ padrão B significa que padrão A precisa do padrão B para completá-lo [3]. Deste modo, o padrão *Task Level Commit* deve ser implementado para que o *Integration Build* funcione.

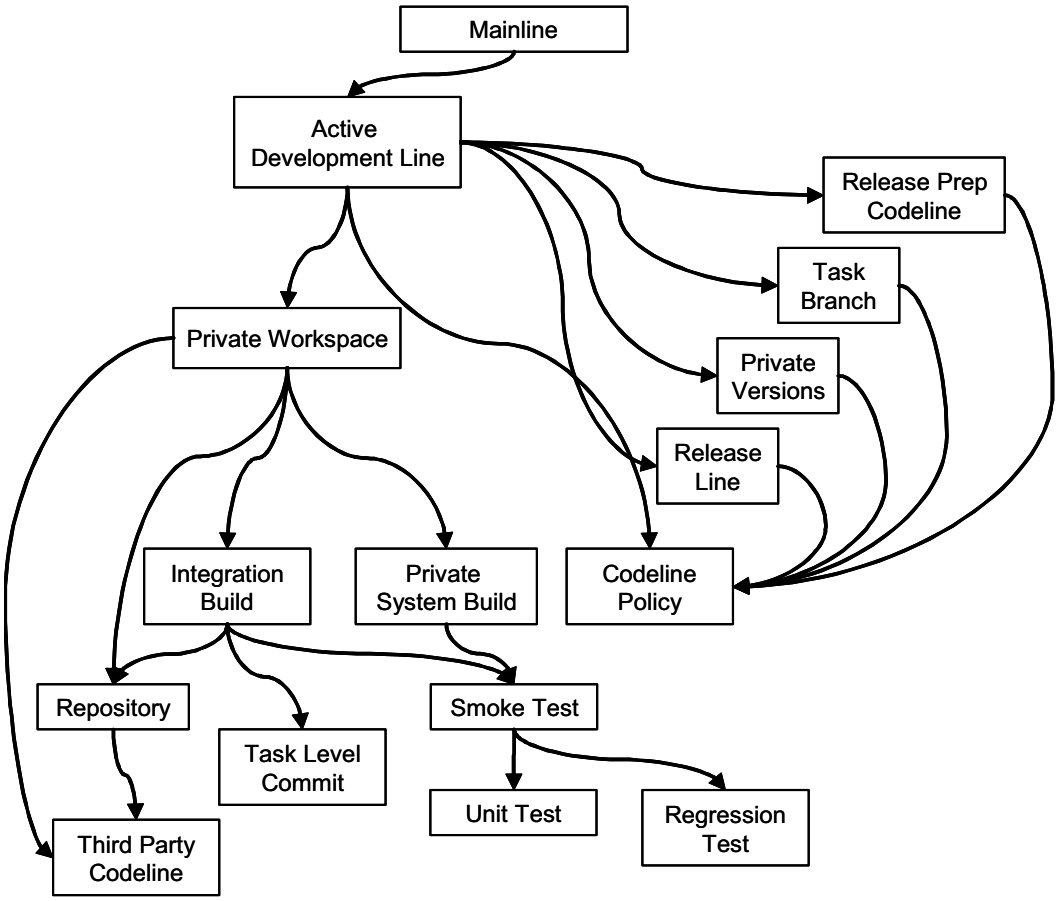


Figura 4. Padrões de gerência de configuração de software [3]

4. Padrões de gerência de configuração de software

Esta seção descreve cada padrão de gerência de configuração de software utilizado no projeto. A aplicação destes padrões é determinada de acordo com as decisões de organização e arquitetura, conforme mencionado na seção 3. Deste modo, alguns padrões de SCM propostos por [3] não foram aplicados e os demais sofreram adaptações considerando o contexto específico do projeto observado.

4.1 Padrão *Mainline*

4.1.1 Contexto de Aplicação

Como mencionado na seção 2, o projeto é desenvolvido por quatro parceiros situados fisicamente em localidades diferentes. Cada parceiro possui sua estrutura organizacional e diferentes tipos de ferramentas para lidar com gerência de configuração. Sendo assim, foram criadas cinco diferentes linhas de codificação (*codelines*), uma para cada parceiro e uma principal gerenciada pela Siemens (figura 5). Além disso, existe uma linha de codificação chamada de *produção* com o propósito de receber somente versões estáveis do software.

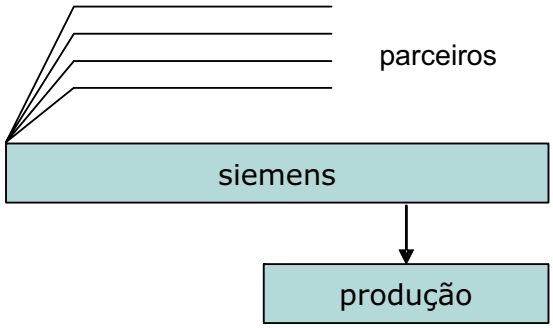


Figura 5. Linhas de codificação

4.1.2 Problemas enfrentados

Alguns componentes possuem dependências entre si, por exemplo, qualquer mudança na interface e/ou comportamento do componente afeta o trabalho de um ou mais parceiros. Se todos os parceiros possuem linhas de codificação diferentes, como resolver este problema de dependência de componentes?

4.1.3 Soluções

Para que fosse possível utilizar cinco diferentes linhas de codificação e ter maior controle de todas as mudanças de interface/comportamento dos componentes, foi desenvolvido um processo de *comunicação de mudança de interface.* Neste processo, o parceiro realiza a mudança na interface/comportamento do componente e depois notifica todos os parceiros afetados.

Esta adaptação é válida, pois, o projeto é norteado por uma arquitetura baseada em componentes e interfaces bem definidas. Ciclos de integração semanais ocorrem na linha principal da Siemens (onde são gerados os *builds* e *releases*) pelos parceiros. Sendo assim, no término de cada ciclo tem-se uma versão definitiva do produto que é disponibilizada na linha de produção para a equipe de teste.

4.1.4 Contexto Resultante

Depois da implementação deste processo, cada parceiro foi capaz de trabalhar em sua própria linha de codificação. O processo de *comunicação de mudança de interface* possibilitou uma melhor comunicação, de acordo com as dependências arquiteturais. Esta comunicação aprimorada permitou uma maior transferência de responsabilidade para os parceiros. O foco da Siemens pode ser mantido no controle da linha de codificação principal. A linha de *produção* (principal), incrementada a partir das integrações semanais, é a única referência – não-ambígua, portanto – para versões oficiais do produto.

4.2 Padrão *Active Development Line*

4.2.1 Contexto de Aplicação

O desenvolvimento ocorre em quatro linhas ativas de codificação, uma para cada parceiro. Isto implica dizer que, para cada uma das linhas, uma partição do sistema (em termos de sub-sistemas e componentes) deve ser desenvolvida, testada e entregue por cada parceiro envolvido no projeto.

4.2.2 Problemas enfrentados

A coordenação das entregas, a fim de garantir um único sistema funcionando não é uma tarefa simples. Muitos problemas de retrabalho ou esforço elevado de integração foram encontrados.

4.2.3 Soluções

Toda integração de código do parceiro é acompanhada por notas de entrega (*delivery notes)*. As notas de entrega têm o propósito de fornecer as condições atuais da entrega, ou seja, quais componentes foram adicionados ou modificados, quais bugs foram resolvidos, quais as limitações de cada componente e assim por diante. O gerente de configuração de software é responsável por revisar as notas de entrega e comunicar aos parceiros eventuais problemas de consistência.

4.2.4 Contexto Resultante

Com o adequado preenchimento das notas de entrega é possível integrar e gerar um novo *build* do sistema e, por conseguinte disponibilizar uma versão estável do mesmo na linha principal de codificação. Esta prática tornou viável a implantação de uma estratégia de *integração por estágios*, na qual um mesmo ciclo de integração é quebrado em ciclos menores de acordo com as dependências entre os componentes.

4.3 Padrão *Private Workspace*

4.3.1 Contexto de Aplicação

Diferentes parceiros possuem diferentes estruturas organizacionais, cultura de trabalho e processos de software. O isolamento geográfico demanda um certo isolamento para a construção do software de modo a eliminar o excesso de interferência no trabalho diário de cada desenvolvedor.

4.3.2 Problemas enfrentados

Com o uso de uma linha de codificação por parceiro, pode-se isolar as mudanças feitas por cada um deles e obter um melhor controle do software sendo desenvolvido. Em cada uma dessas linhas, cada desenvolvedor de cada parceiro pode gerar o seu espaço de trabalho (*workspace*).

Mesmo usando diferentes linhas de codificação, pôde ser observada a utilização de forma intrusiva, isto é, parceiro P2 modifica componentes atribuídos ao parceiro P1 (conforme mostrado na figura 6). Este problema causa desperdício de esforço visto que não há garantia de consistência nas versões produzidas por P1.

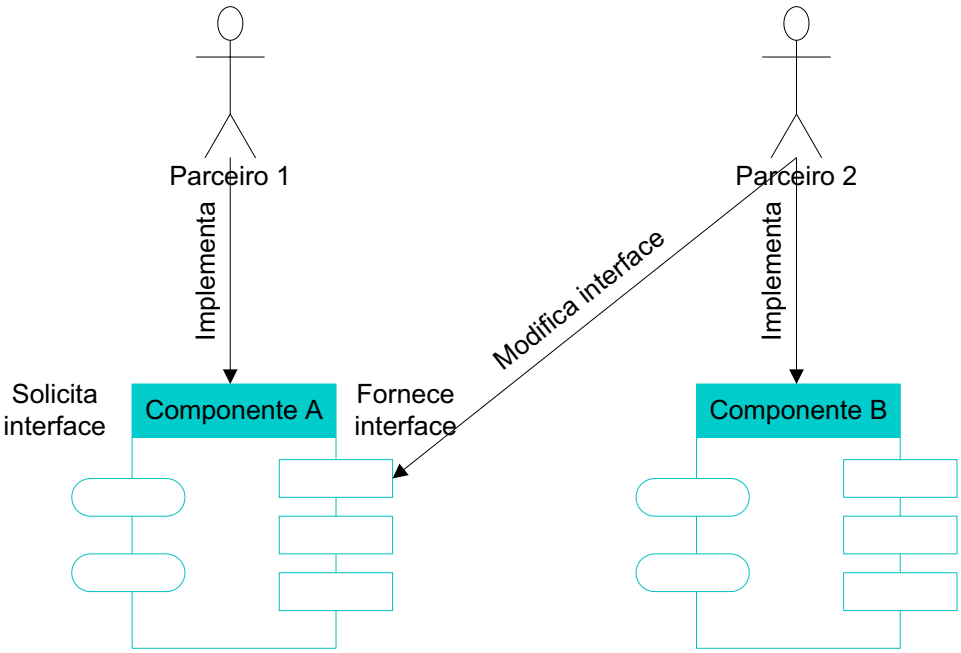


Figura 6. Mudança de interface de componentes

4.3.3 Soluções

O estabelecimento de um processo de comunicação de mudança de interface e a separação de responsabilidades atenuou os problemas enfrentados. A arquitetura bem definida e o reforço da atribuição das responsabilidades desempenharam um papel fundamental na redução deste tipo de problema.

4.3.4 Contexto Resultante

Melhoria na comunicação entre os times e na integração dos componentes implicando em *builds* mais estáveis e entregas mais controladas.

4.4 Padrão *Integration Build*

4.4.1 Contexto de Aplicação

Uma data e horário do dia são marcados para cada parceiro realizar sua entrega. Quatro entregas são realizadas na semana, de acordo com as dependências – *integração por estágios*. Esta abordagem favorece a integração gradual do produto. Cada uma das integrações deve ser acompanhada por notas de entrega e um rótulo (*tag)* atribuído à versão específica dos componentes sendo entregues (conforme mostrado na figura 7). As notas de entrega devidamente preenchidas e a *tag* associada aos componentes facilitam a integração e a geração de um novo *build* do sistema.

4.4.2 Problemas enfrentados

Algumas integrações não foram acompanhadas de notas de entrega devidamente preenchidas, o que dificultou a implementação da *integração por estágios*. Além disso, a *tag* era associada a todos os componentes do sistema, dificultando a assimilação do que estava sendo entregue de fato.

4.4.3 Soluções

Para cada integração realizada pelo parceiro, o gerente de configuração de software é responsável por verificar se a *tag* foi corretamente atribuída aos devidos componentes e checar se as notas de entrega condizem com o conjunto de componentes entregues. Com estas informações em mãos, o SCM comunica aos parceiros da inconsistência da entrega. Atualmente, um *checklist* é aplicado a fim de validar e fornecer *feedback* de cada entrega.

4.4.4 Contexto Resultante

*Build*s de integração são produzidas de forma sistemática, com periodicidade definida e controlada. No caso do projeto em questão, usou-se a freqüência semanal.

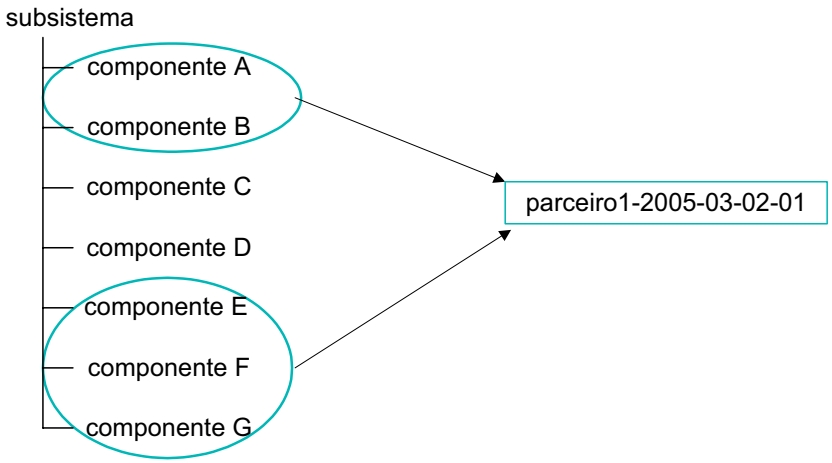


Figura 7. Parceiro 1 atribuindo uma *tag* aos componentes

4.5 Padrão *Third Party Codeline*

4.5.1 Contexto de Aplicação

Cada parceiro possui uma linha de codificação no sistema de controle de versão. O ciclo de vida da linha de codificação (atualização, desenvolvimento, teste de integração interna e assim por diante) está sob responsabilidade do parceiro. As atualizações bem como as entregas devem ocorrer de acordo com políticas bem estabelecidas no início do projeto.

4.5.2 Problemas enfrentados

Má gestão das linhas de codificações dedicadas aos parceiros. Muita demanda interna para integrar e gerar builds, o que afeta diretamente o caminho crítico do projeto.

4.5.3 Soluções

Papéis e responsabilidades foram bem definidos e enfatizados para cada parceiro envolvido. Neste sentido, foram desenvolvidos, documentados e divulgados todos os procedimentos e políticas de gerência de configuração.

4.5.4 Contexto Resultante

Seguindo todos os procedimentos estabelecidos pela Siemens, cada parceiro foi capaz de cuidar, com certo grau de independência, do ciclo de vida da sua linha de codificação. Boa parte do caminho crítico do projeto foi aliviada, graças a este redirecionamento de responsabilidade.

4.6 Padrão *Task Level Commit*

4.6.1 Contexto de Aplicação

Uma tarefa (ou *task*), para o projeto analisado, pode ser mapeada em uma entrega individual por parceiro. Por exemplo, implementar um novo componente ou serviço. Cada parceiro realiza a sua entrega em uma linha de codificação isolada, usando controle de versão a partir de *tag*.

4.6.2 Problemas enfrentados

Cada parceiro é responsável por tarefas e pelo ciclo de vida de sua linha de codificação individual. Em alguns momentos, quando um parceiro depende fortemente da modificação de outro, o dependente terá que aguardar pela conclusão da tarefa, o que pode significar tempo improdutivo de espera.

4.6.3 Soluções

Isolamento das linhas de codificação por parceiro e a entrega baseada em *tag*. Para contornar o problema da espera por dependência, usa-se o conceito de *snapshot*, ou fotografia. Nesta abordagem, o parceiro causador da dependência prioriza as suas atividades e, antes da conclusão da tarefa, aplica um rótulo em sua linha de codificação tornando a versão disponível aos demais interessados. Uso intensivo de notas de entrega com o propósito de comunicar efetivamente o andamento das modificações para todos os envolvidos.

4.6.4 Contexto Resultante

Percepção consistente do andamento do projeto pelos líderes de projeto, devido ao incremento consistente de funcionalidades no software. Maior controle sobre os *builds* globais do produto em construção a partir de diferentes entregas. Necessidade de pessoas dedicadas à tarefa de monitorar o andamento das modificações e integrar componentes.

4.7 Padrão *Smoke Test*

4.7.1 Contexto de Aplicação

Mesmo entregas bem estruturadas, tal como proposto em *Task Level Commit,* demandam uma verificação mínima de consistência. Eventualmente versões são enviadas para outros times remotamente localizados ao redor do globo a fim de executar testes de diversos tipos.

4.7.2 Problemas enfrentados

Contínuo balanceamento entre custo, *time-to-market* e estabilidade de cada *build* (figura 8). Difícil decisão entre lançar um *build* antes ou depois dos *Smoke Tests*.

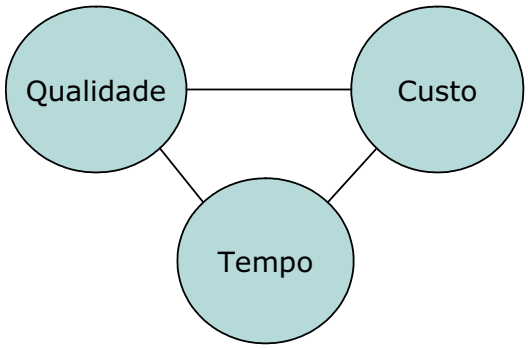


Figura 8. Triângulo mágico [19]

4.7.3 Soluções

Definir funcionalidades prioritárias para a execução de *smoke tests*. Estratégia de priorização com base nos componentes que sofreram as mudanças mais críticas desde a última entrega.

4.7.4 Contexto Resultante

Mais tranqüilidade e confiança antes de repassar versões intermediárias do software para diferentes times de teste geograficamente distribuídos. Menos desperdício de esforço e redução na demanda por comunicação entre os times.

4.8 Padrão *Regression Test*

4.8.1 Contexto de Aplicação

Defeitos críticos são encontrados e corrigidos no software. Dependendo de quão críticos, há uma necessidade de garantir que os mesmos não voltarão a se manifestar em uma nova versão.

4.8.2 Problemas enfrentados

Definição de prioridades dos defeitos encontrados no software. Muita comunicação entre times de desenvolvimento no projeto com o propósito de rastrear tais defeitos e propor soluções para os mesmos.

4.8.3 Soluções

Criação do papel do gerente de versão intermediária (*Build Manager*) para cada um dos parceiros. O BM é responsável por fazer o rastreamento dos *bugs* sob sua responsabilidade e notificar (com a ajuda da equipe de teste), para cada entrega, quais as correções e pendências. Além disso, o BM e arquiteto devem fornecer soluções e prazos para os defeitos que se manifestam em uma nova versão do software.

4.8.4 Contexto Resultante

Áreas funcionais de alta prioridade estão protegidas pelos testes. Entretanto, há uma demanda por comunicação para definir qual o nível de regressão desejado em cada execução de testes.

4.9 Padrão *Release Line*

4.9.1 Contexto de Aplicação

Necessidade de ter uma base única para a geração de versões oficiais do produto. Linhas de codificação dos parceiros são isoladas da linha de codificação de produção, responsável pela geração de *releases oficiais do produto*.

4.9.2 Problemas enfrentados

Esforço para a junção das linhas de codificação dos diversos parceiros (processo de integração) e necessidade de verificar a consistência de cada entrega através de um *checklist* desenvolvido pelo gerente de configuração de software. Este *checklist* tem como finalidade a aceitação ou rejeição da entrega do parceiro.

4.9.3 Soluções

A própria linha de codificação de produção é utilizada como linha para geração de um novo release do produto, evitando o uso de uma linha de codificação.

4.9.4 Contexto Resultante

Capacidade de produzir *releases* do software independentemente de qualquer tarefa de desenvolvimento em andamento pelos parceiros.

4.10 Padrão *Codeline Policy*

4.10.1 Contexto de Aplicação

Diferentes organizações, diferentes culturas, todos envolvidos na construção de um único software. Necessidade de um nível mínimo de uniformidade nas ações a fim de garantir a produtividade coletiva dos desenvolvedores.

4.10.2 Problemas enfrentados

Dificuldade para os desenvolvedores assimilarem todas as idéias contidas na política de gerência da linha de codificação. Divergências entre formas de trabalho, cultura e metodologia de desenvolvimento de software.

4.10.3 Soluções

Os parceiros receberam um treinamento de gerência de configuração de software no início do projeto. Desta forma, foram definidos e enfatizados os papéis e responsabilidades no processo de SCM. Cada parceiro é responsável pelo ciclo de vida de sua respectiva linha de codificação (atualizações, *commits*, rotulação e entregas oficiais). Concentração das responsabilidades mencionadas no papel do *Build Manager*. Cada parceiro possui um desenvolvedor desempenhando o papel especial de BM.

4.10.4 Contexto Resultante

Redução da sobrecarga e dos ruídos na comunicação no grupo. Agilidade na tomada de decisão a cada ciclo de integração. Facilidade para absorver novos desenvolvedores no projeto. Maior potencial para que todas as linhas de codificação se mantenham mais estáveis ao longo do projeto.

5 Conclusão

Um projeto desenvolvido em regime de terceirização, envolvendo equipes de diferentes localidades e de diferentes empresas, influenciou fortemente a estruturação das soluções de gerência de configuração para o projeto observado. A necessidade de compartilhar artefatos de software em um contexto global de países e organizações colocou grandes desafios para o projeto. As diferenças culturais e de processo de software também contribuem fortemente com a complexidade do ambiente analisado.

As soluções foram sendo implementadas de acordo com as necessidades do projeto, gradualmente, bem como o reconhecimento dos padrões proposta por [3]. Deste modo, foi possível identificar os padrões e traçar as devidas correlações com o ambiente real. Neste âmbito, a linguagem contribuiu para a reflexão e validação das soluções vigentes. Considerando fatores como organização, arquitetura e *time-to-market*, pode-se também utilizar outras linguagens de padrão, como por exemplo, os padrões organizacionais definidos por [2]. Desta forma, é possível compreender de forma ampla o ciclo de influência entre organização, arquitetura e gerência de configuração.

6 Apêndice "A" – Ferramentas utilizadas

Este apêndice contém informações sobre as ferramentas utilizadas pelo projeto para dar sustentação ao emprego dos padrões de gerência de configuração. As informações estão contidas na tabela 1.

Tabela 2. Ferramentas empregadas na aplicação dos padrões de SCM

Ferramenta	Aplicação
Subversion	Repositório de versões. Criação de branches (linhas de codificação) para permitir o trabalho simultâneo dos diversos parceiros e posterior integração [8].
Kdiff3	Checar se ocorre sobreposição entre o código entregue pelos parceiros. Permite comparar diretamente três fontes de dados [9].
Ant/ make	Automação de tarefas envolvendo o produto do software e os espaços de trabalho (workspace) [10]/[11]: • Geração de builds • Empacotamento de componentes • Instrumentação de código para *profiling* (análise dinâmica) • Invocação das ferramentas para a análise estática do código
Check	Infra-estrutura C para execução de testes automatizados escritos na mesma linguagem [12].
Checkstyle	Análise de padrões de codificação. Detecção de práticas perigoras (*anti-patterns*) [13].
CCCC	Métricas de código: tamanho de módulos e funções em NCSS

	(*non-commented source statement*), complexidade ciclomática por função. Suporte à C/C++ [14].
JavaNCSS	Análogo ao CCCC, só que para Java [15].
Simian	Analisador estático de redundância no código. Suporta múltiplas linguagens: C/C++, Java, C#, entre outras [16].
RPM/ JAR	Padrões para empacotamento de componentes e produtos de software [17]/[18].
Unix tools	grep, sed, find, bash scripts, etc.

7 Referências

[1] Osterweil, L., *Software Processes are Software Too,* ACM, 1987.

[2] Coplien, J., Harrison N., *Organizational Patterns for Agile Software Development*, Prentice Hall, 2004.

[3] Berczuk, S., Appleton, B., *Software Configuration Management Patterns,* Addison-Wesley, 2002.

[4] Bass, L., Kazman, R., Clements, P., *Software Architecture In Practice*, SEI Series in Software Engineering, 2002

[5] McConnel, S., Rapid Development, Microsoft Press, 1996

[6] Fowler, M., *Continuous Integration*, http://www.martinfowler.com/articles/continuousIntegration.html, última visita em [09/07/2005].

[7] Gabriel, R., *Software Patterns*, Oxford Press, 1996

[8] Subversion, http://subversion.tigris.org/, última visita em [10/07/2005].

[9] Kdiff3, http://kdiff3.sourceforge.net/, última visita em [10/07/2005].

[10] Apache Ant, http://ant.apache.org/, última visita em [10/07/2005].

[11] GNU make, http://directory.fsf.org/make.html, última visita em [10/07/2005].

[12] Check, http://check.sourceforge.net/, última visita em [10/07/2005].

[13] Checkstyle, http://checkstyle.sourceforge.net/, última visita em [10/07/2005].

[14] CCCC, http://cccc.sourceforge.net/, última visita em [10/06/2005].

[15] JavaNCSS, http://www.kclee.de/clemens/java/javancss/, última visita em [10/06/2005].

[16] Simian, http://www.redhillconsulting.com.au/products/simian/, última visita em [10/06/2005].

[17] RPM, http://www.rpm.org/, última visita em [10/06/2005].

[18] JAR, http://java.sun.com/j2se/1.5.0/docs/guide/jar/, última visita em [10/06/2005].

[19] Göhner, P. (2003). *Lecture notes of Software Engineering for Real-Time Systems*. IAS, Stuttgart.

Extending Patterns with Testing Implementation *

Maria Istela Cagnin[†1], Rosana T. V. Braga[1], Fernão S. Germano[1], Alessandra Chan[‡1], José Carlos Maldonado[1]

[1]Laboratório de Engenharia de Software
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
Av. do Trabalhador São-Carlense, 400 – 13560-970 São Carlos, SP

{istela, rtvb, fernao, alechan, jcmaldon}@icmc.usp.br

Abstract. *VV&T activities are a present concern in the context of patterns, as patterns are used for software development, maintenance, and reengineering, and VV&T is necessary to assure the quality both of the solutions and of the delivered products. Although VV&T activities are important, they are not always performed as they should be, due to the associated time and cost. In this context, this paper proposes a strategy that allocated test resources to software patterns. This allows reusing not only solutions in a certain context, but also the corresponding test resources needed to validate applications. Reengineering case studies were conducted with the support of a pattern language, through which it has been possible to observe, although without statistical significance, a meaningful reduction of the time spent with VV&T activities when test resources are allocated to patterns.*

1. Introduction

Software patterns are being widely used to enhance productivity. Besides providing solutions to recurring problems, they embed knowledge and the experience of experts in a domain. Software patterns are used at several abstraction levels: analysis patterns (Coad, 1992; Coad et al., 1997; Larman, 2004), design patterns (Gamma et al., 1995; Larman, 2004), architectural patterns (Beck and Johnson, 1994; Buschmann et al., 1996), testing patterns (Binder, 1999; DeLano and Rising, 1998), reverse engineering patterns (Demeyer et al., 2000), reengineering patterns (Stevens and Pooley, 1998; Recchia, 2002; Lemos, 2002), among others.

More specifically, testing patterns provide general guidelines and procedures to help testers during product quality assessment, but do not capture the expert solution nor the specific validation aspects of the applications. A good solution should have the corresponding validation captured in the pattern. This allows the reuse of VV&T (Verification, Validation and Test) information, in addition to the reuse of the solutions.

Several authors (Sommerville, 2000; Harrold, 2000; Pressman, 2001; Rocha et al., 2001) comment on the importance of testing activities, although they point out that these activities are not always practiced due to the associated time and cost. Testing resources, ready to be used, allocated to software patterns, will ease the execution of testing activities

†Financial support from FAPESP #00/10881-4.

‡Financial support from CNPq.

and, consequently, will make it possible to enhance the quality of delivered products. Moreover, this can stimulate the execution of tests before coding, in the context of agile methods (Beck et al., 2001). This way of performing testing activities is called e**X**treme **T**esting (XT) by Myers (2004) and *Test-Driven Development* (TDD) by Beck (2002). The latter is also known as *Test-First Development* (Larman, 2004). Test creation before coding requires understanding the specification and the reduction of ambiguity before coding begins.

This paper proposes considering VV&T activities in any type of software pattern definition. For that, it proposes a strategy that allocates test resources (requirements, test cases and other produced documents) to software patterns, through the inclusion of a new pattern section. As patterns can be used in software development, maintenance, and reengineering, and testing activities are included in the quality assessment of any product delivered to users (Rocha et al., 2001), patterns should be concerned with VV&T activities.

Two reengineering case studies were conducted with the support of an agile reengineering process, called PARFAIT [1] (Cagnin et al., 2003b), that uses the GREN framework (Braga and Masiero, 2002), which was built based on the GRN pattern language (Braga et al., 1999) and belongs to the business resource management domain. In one of the case studies a pattern language with allocated test requirements was used, and it was possible to observe the reduction of both time and effort during software reengineering. That is due to the fact that a large percentage of the test cases were based on the reuse of the test requirements available in the patterns definition.

This paper is organized as follows: in Section 2 related work in the context of this paper is presented. In Section 3 a strategy that allocates test resources to software patterns is described. In Section 4 we describe an experience using the proposed strategy in the patterns of the GRN pattern language, generating a specific strategy. In Section 5 the results of two reengineering case studies are presented to compare the advantage of the proposed strategy. In Section 6 conclusions are presented and future work is discussed.

2. Related Work

Tsai et al. (1999) discuss tests in design patterns in object oriented frameworks. They present a technique, called *Message Framework Sequence Specifications (MfSS)*, to support template scenario generation used to create types of test scenarios, such as partition test scenarios and random test scenarios. These scenarios support the test of applications generated from object oriented frameworks that use extensible design patterns, i.e., patterns that allow the addition of new classes and methods to the framework at compilation or runtime. *MfSS* is an extension of the techniques *Method Sequence Specifications (MtSS)*, and *Message Sequence Specifications (MgSS)*, which specify the message interaction among objects of object oriented applications, through regular expressions.

Weyuker (1998) argues that it is important to make available, with software components and their specification, the test cases and the parts of the component that are exercised with these test cases. The reason is that components need to be tested in each new environment as they interact with other components. However, Mariani et al. (2004) observe that components are being used in ways not anticipated by their developers, making their specifications invalid, as well as their available test sets. Mariani et al. (2004)

[1] PARFAIT - from the Portuguese "**P**rocesso **Á**gil de **R**eengenharia baseado em **Fr**A**m**ework no domínio de sistemas de **I**nformação com técnicas VV&**T**", which means: An Agile Reengineering Process based on an Information System Domain Framework using VV&T Activities.

propose an approach for implementing self-testing components, which allows integration test specifications and suites to be developed by observing both the behavior of the component and of the entire system. Self-testing components can self-verify their behavior in the new context and, thus, in principle, can be reused without any a-priori limitation, so they are provided with a set of associated test cases that are executed at system deployment time.

Tevanlinna et al. (2004) present several testing approaches applied in the context of product families. (Clements and Northrop, 2001). In one of them, called *reusable asset instantiation* (McGregor, 2001) **apud** (Tevanlinna et al., 2004), the test assets are created as extensively as possible in domain engineering, anticipating variabilities by creating, for example, document templates and abstract test cases. In application engineering, a full testing process according to the levels of the V-model [2] (Germany Ministry of Defense, 1992) (acceptance test, system test, integration test and unit test) is instantiated. The concrete test assets are used as is, and the abstract assets are extended or refined to test the product-specific aspects in application engineering (McGregor, 2001). Product family testing differs from traditional testing in the reuse not only of resources related to the architecture and components, but also testing resources. It is important to distinguish between testing resources that belong to the domain and those that belong specifically to the product. However, the literature lacks testing methods specific to product families, which should focus primarily on unit and integration test.

As mentioned in the previous section, test resources allocated to patterns and components, product families, and frameworks can support the practice of *Test-Driven Development*. Larman (2004) mentions several advantages of this practice, among which are: **the unit tests actually get written:** unit test writing is not often done if it is left as for later; **provable, repeatable, automated verification:** having hundreds or thousands of unit tests built in a test tool, during weeks, allows meaningful verification of system correctness; and **the confidence to change things:** if a change was made to the system, the unit test set for the modified classes needs to be executed to know if the change caused any errors.

3. Patterns with Testing

The strategy proposed in this paper allocates test resources to software patterns. This strategy was motivated by the importance of easing the application of VV&T activities in the production of software based on patterns, as mentioned in Section 1. The steps of the proposed strategy are presented in Table 1. In the specific case of analysis and design patterns, each element that participates in the pattern is equivalent to the classes that compose the structure of such patterns.

In step **Define types of requirements** it is necessary to identify the requirement types, according to the software aspects that should be considered by the tests, as they are important to system verification and validation. This depends on the domain and on the context for the pattern. For example, analysis patterns in the domain of business resource management require tests related to consistency, integrity, and business rules. In this case, integrity should be considered because a large volume of data is supplied by users, and input errors in these data should be avoided. Consistency should be considered because data needs to be physically stored, for example using a relational or object-oriented database, so it is necessary to guarantee that the data will be retrieved successfully later on. Business rules should also be considered so that the system correctly satisfies organization business rules. Another example of a requirement type is for reengineering

[2] V-model of software testing that is the traditional way to model testing.

Table 1: Steps of the strategy for aggregating test resources to software patterns

– Given a software pattern

- **Define** types of requirements
- **Select** existing test criteria
- **For** each type of requirement defined and common to most pattern elements
  - **Create** test requirements based on the selected test criteria
- **For** each pattern element
  - **For** each test requirement type defined and specific to the pattern element
    - ∗ **Create** test requirements based on the selected test criteria

– Classify and document each test requirement created
– Derive test cases
– Map common test cases
– Make the test resources available

patterns, where it is relevant to worry about tests that ensure functional compatibility with the legacy system. For design patterns, it should be important to test, for example, system flexibility.

In step **Select existing test criteria**, it is necessary to decide which existing test criteria will be used. A test criterion aims at selecting and evaluating test cases to increase the chances of revealing defects or, when this does not occur, to establish a high confidence level in product correctness (Rocha et al., 2001).

Each test criterion contributes a set of specific test cases, but any one of them offers a complete set. Thus, test strategies need to be established, containing both functional and structural criteria, so that it is possible to achieve a complete test set. According to Myers (2004) and Roper (1994), functional and structural test criteria should be used together so that one complements the other.

Initially, the type of criterion to be used has to be studied. Functional test criteria should be selected when the product specification is considered to derive the test requirements, for example, when the pattern is an analysis pattern. On the other hand, when product implementation is considered, structural test criteria can be selected, for example when the pattern is a design or implementation pattern. After selecting the type of criteria, observe the goals, cost, efficiency and strength [3] of the criteria belonging to the selected type, so that they can be correctly chosen.

Functional test criteria “equivalence partitioning” and “boundary value analysis” were used in the experience of allocating tests to analysis patterns (see next Section), as they are the most common in the literature. The “equivalence partitioning” criterion divides the input domain of a program into a finite number of equivalence classes (both valid and invalid), and derives test cases from these classes. This criterion aims at minimizing the number of test cases, by selecting at most one test case for each equivalence class, as, in principle, all elements of a class should behave in an equivalent way (Rocha et al., 2001). In other words, if a test case for a given equivalent class reveals an error, any

[3] or satisfaction difficulty, which refers to the difficulty of satisfying the criteria after having already satisfied another (Rocha et al., 2001).

other test case of this class should reveal the same error (Myers, 2004). The functional test criterion "boundary value analysis" complements the "equivalence partitioning" criterion and is concerned with creating test cases that consider values directly below or above the bounds of equivalent classes. According to Myers (2004), test cases that explore boundary conditions are more worthwhile than those that do not.

In step **Create test requirements based on the selected test criteria** test requirements for the types defined are created, based on the guidelines of each selected test criterion. Each test requirement should be classified and documented (step **Classify and document each test requirement created**). The test requirement classification is related to "when" the test should be executed. For example, in analysis patterns of the business resource domain, the execution of a test is done during the data manipulation operations (i.e., insertion, modification, deletion, and search). In design patterns, the test execution can be done before (in the design models that are built) or after the system implementation (in the source code that is created). The documentation should supply the information needed to support the next step (**Derive Test Cases**), considering the test requirements' specification and valid conditions.

In step **Map common test cases**, the test cases that are common to most pattern elements are mapped to the patterns that should use them, to encourage reuse. The mapping documentation should contain the following information: **pattern name**, **pattern class**, **test case number**, and **previous validations**.

In step **Make the test resources available**, both test requirements and test cases are included in the pattern documentation, with the introduction of a new section called *VV&T Information*. In this section, the test criteria used in the test resources creation should be mentioned. In the case of pattern languages, systems of patterns or any other types of pattern collections, test resources that can be used by all patterns should be placed in a general *VV&T Information* section that could be shared by all of them, while the test resources that are specific to each individual pattern and the test cases mapped in step **Map common test cases** are placed in its own *VV&T Information* section. For example, in the GRN pattern language, the test resources relative to business rules together with the mapping done in step **Map common test cases** are placed in the *VV&T Information* sections corresponding to the patterns to which they belong, while the test resources corresponding to consistency and integrity are placed on a general *VV&T Information* section, because they are applicable to all patterns.

4. Analysis Patterns with Testing: An Experience

The strategy presented in the previous section to allocate test resources to software patterns was used in several analysis patterns that compose the GRN pattern language, for the business resource management domain (Braga et al., 1999). GRN has fifteen analysis patterns, some of which are applications or extensions of existing patterns in the literature.

Figure 1 presents GRN patterns, the dependencies among them, and the order in which they can be applied. This pattern language has three main patterns: RENT THE RESOURCE (4), TRADE THE RESOURCE (6), and MAINTAIN THE RESOURCE (9). The application of each of these patterns and, consequently, of the patterns associated with them, is done according to the goal of the application being modeled. Their use is not mutually exclusive, as there are applications in which they can be used in parallel (for example, in a car repair shop that also sells and buys parts, in addition to repairing cars).

According to Figure 1, the patterns are categorized in three groups, depending on their goal. Group 1 (Identification of the Business Resource) has three patterns, (1),

(2) and (3), which are concerned with the identification and possible qualification, quantification and storage of the resources managed by the organization. Group 2 (Business Transactions), has seven patterns, (4) to (10), which are concerned with the operations performed with the resources by the application. Group 3 (Business Transaction Details) contains five patterns, (11) to (15), which are concerned with the details of the transactions performed with the resource.

Initially, during the execution of step **Define types of requirements** of the strategy, three types of test requirements were identified: consistency, integrity, and business.

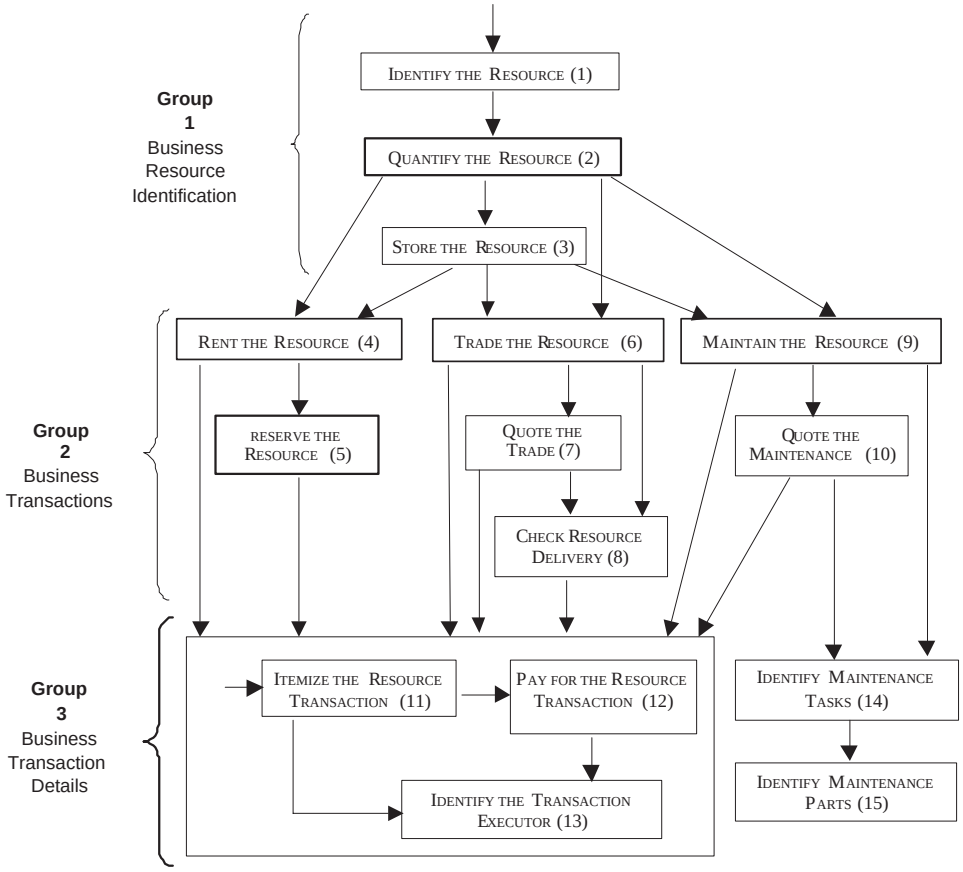


Figure 1: Structure of the GRN pattern language (Braga et al., 1999)

The definition of the **consistency** test requirement was motivated by the importance of validating the data submitted to the system before they are stored. The definition of the **integrity** test requirement was motivated by the importance of correct storage of the data processed by the information system in a relational database and, mainly, by the need to ensure the integrity of the stored data, so that they are later correctly retrieved. The **business** test requirement has the goal of assessing the correct treatment of the business rules and ensuring that the system works properly. This requirement is based on specific business features, and concerns the business functions embedded in the pattern.

As GRN analysis patterns provide knowledge about the domain functionality, in the form of solutions to analysis problems, rather than source code, only the functional test criteria were considered during the execution of step **Select existing test criteria**. As discussed in the previous section, the "equivalence partitioning" and "boundary value analysis" criteria (Myers, 2004) were selected, in addition to the ideas behind the "Input Validation Test" technique [4] (Hayes and Offutt, 1999).

[4]the "Input Validation Test" technique identifies test data that try to show the presence or lack of specific

Table 2 presents the steps of a strategy to allocate test resources to GRN patterns, based on the strategy proposed in Section 3. Most steps were maintained as a one to one relationship, except by step **Create test requirements based on the selected test criteria**, which was split into three steps (Create consistency test requirements, Create integrity test requirements and Create business test requirements).

Table 2: Strategy to allocate test resources to GRN patterns

– **Define** types of requirements
– **Select** existing test criteria
– **Create** consistency test requirements
– **Create** integrity test requirements
– **For** each GRN pattern

- **For** each pattern class
  - **Create** business test requirements

– **Classify and document** each of the test requirements
– **Derive** test cases
– **Map** common test cases
– **Make** the test resources available

Consistency and integrity test requirements that are common to most GRN pattern classes (step **Create consistency test requirements** and step **Create integrity test requirements**) were defined based on the equivalence classes created by applying the selected functional test criteria. The documentation of the equivalence classes was adapted from the one suggested by (Myers, 2004) and is presented in Table 3.

Table 3: Global Equivalence Classes

Operation	Requirement Type	Valid Classes	Invalid Classes
Include, Modify, Delete	consistency checking	integer attribute **(1)**	non integer attribute**(2)**
Include, Modify, Delete	consistency checking	attribute value between 1 and 2147483647 **(3)**	attribute value < 1 **(4)**
			value > 2147483647 **(5)**
Include, Modify, Delete	consistency checking	attribute value between 0 and 2147483647 **(6)**	attribute value < 0 **(7)**
Include, Modify, Delete	consistency and integrity checking	filled attribute value **(8)**	empty attribute value **(9)**
Include, Modify	consistency checking	alphanumeric attribute value**(10)**	non-alphanumeric attribute value **(11)**
...	...	...	...
Include, Modify	integrity checking	attribute is foreign key registered as a primary key in the associated table **(47)**	attribute is foreign key and is not registered as a primary key in the associated table **(48)**
Delete	integrity checking	attribute without pending relationships (it is not a foreign key) **(49)**	attribute with pending relationships **(50)**

To create the global consistency equivalence classes, both valid and invalid equivalence classes have to be considered for each primitive data type (`integer`,

faults, concerning "input tolerance" (i.e. verifies the system ability to adequately processing input values, both expected or non-expected).

`float`, `string` and `date`); as well as for other data types (for example, `vector`, `enumeration`, `multivalue`, etc), which are used by the analysis pattern language. For attributes whose type is different from the primitive ones, it is necessary to check the possible valid and invalid equivalence classes, which can be abstracted from the suggestions established in (Cagnin et al., 2004) for the types `integer`, `float`, `string`, and `date`, and others can be created to consider specific characteristics of the attribute type.

To create the global integrity equivalence classes, both valid and invalid equivalence classes should be created for each integrity rule of the relational database. In this work, the integrity rules are limited to primary and foreign keys of relational databases, so other features such as *triggers* and *stored procedures* are not considered.

In step **Create business test requirements** valid and invalid equivalence classes are created from specific business functions embedded in the patterns of the analysis pattern language (business rules of the domain for which the analysis pattern languages belongs), taking into account its conditions. For example, in a rental, the resource can only be rented if it is available at the moment. The documentation of the equivalence classes for the business test requirement was also adapted from that proposed by (Myers, 2004) and contains additional information compared to the one presented in Table 3: the **pattern class** to which the requirement belongs, the corresponding **table** in the RDBMS, the **attribute** of the pattern class that is involved in the business rule and a **comment** about when the test requirement should be considered, if necessary. In Table 4 an example is given to illustrate the documentation of the equivalence class for this type of requirement.

Table 4: Equivalence Class for the business test requirement of the "RENT THE RESOURCE" class

Operation	Pattern Class	Corresponding Table	Attribute	Valid Classes	Invalid Classes	Comment
Include	Rent the Resource	ResourceRental	situation of the resource instance	situation of the resource instance is "available" **(51)**	situation of the resource instance is "unavailable" **(52)**	Requirement should be considered only if the "Itemize the Resource Transaction" pattern was not used and if the "Instantiable Resource" sub-pattern was used.
...	...	...	...	...	...	...

Notice that the creation of the test requirements , in addition to being based on the valid and invalid equivalence classes, are also based on the "boundary value analysis" criterion, by observing the bounds immediately above and below each equivalence class. In step **Create business test requirements**, business test requirements were created for each participating class of each pattern of the GRN pattern language.

All the requirements defined for GRN are classified in step **Classify and document each test requirement created**, according to the data manipulation operation (include, modify, delete or search) done in the relational database. This occurs because GRN was used as basis for the construction of the GREN framework (Braga and Masiero, 2002), using MySQL (MySQL, 2003) RDBMS, which is a relational database.

Another activity of step **Classify and document each test requirement created** is to document all the requirement types created. To support that, Cagnin et al. (2004) suggests that the documentation be presented in a tabular format. The documentation of the test requirements that are common to most patterns should contain the following information: test requirement **number**; test requirement **type** (e.g, integrity, persistence, or business); type of data manipulation **operation** treated by the test requirement (e.g.,

include, modify, delete, or search); **equivalence classes numbers** used to create the test requirement; **test requirement specification**, describing what should be considered by the input data of the test case to be instantiated in **Derive test cases**; **valid condition** to be considered when analyzing the input data and establishing the expected return; **previous validation**, i.e., test requirements that are considered as pre-condition of the test requirement being documented; **comment**, which contains some relevant information regarding the test requirement; and **expected return**. A portion of the test requirements documentation that is common to most GRN patterns, created from the global equivalence classes illustrated in Table 3, is shown in Table 5.

Table 5: Partial Documentation for the consistency and integrity test requirements of GRN patterns

Test Req Number	Type	Operation	Equiv Classes	Test Req Spec	Valid Conditions	Previous validation	Comment	Expected Return
TR01	Consist.	Include, Modify, Delete, Search	2	non integer attribute	integer attribute	–	–	consistency verification error, operation cannot proceed
TR02	Consist.	Include, Modify, Delete, Search	1,4,8	attribute == 0	attribute has an integer value (greater than 0 and less than 2147483648)	–	–	consistency verification error, operation cannot proceed
TR03	Consist.	Include, Modify, Delete, Search	1,3,8	attribute == 1	attribute has an integer value (greater than 0 and less than 2147483648)	–	–	well succeeded verification, operation can proceed
TR04	Consist.	Include, Modify, Delete, Search	1,3,8	attribute == 2147483647	attribute has an integer value (greater than 0 and less than 2147483648)	–	–	well succeeded verification, operation can proceed
TR05	Consist.	Include, Modify, Delete, Search	2,5,8	attribute > 2147483647	attribute has an integer value (greater than 0 and less than 2147483648)	–	–	consistency verification error, operation cannot proceed
TR06	Consist.	Include, Modify, Delete, Search	1,4,8	attribute < 1	attribute has an integer value (greater than 0 and less than 2147483648)	–	–	consistency verification error, operation cannot proceed
TR07	Consist.	Include, Modify, Delete, Search	9	empty attribute	attribute is not empty	–	–	consistency verification error, operation cannot proceed
...	...	...	...	...	...	...	...	...
TR08	Integr.	Include	44	register is logged	register is not logged	TR01 to TR07	–	integrity verification error, operation cannot proceed
TR09	Integr.	Modify, Delete, Search	46	register is not logged	register is logged	TR01 to TR07	–	integrity verification error, operation cannot proceed
...	...	...	...	...	...	...	...	...

To document the business test requirements that are specific to each class of the GRN patterns, the following information is added: **pattern name**; **class name** and **RDBMS table**. A portion of the documentation for the business test requirement of GRN pattern 4 (RENT THE RESOURCE) is presented in Table 6.

Guidelines to support the consistency (step **Create consistency test requirements**) and integrity (step **Create integrity test requirements**) test requirements creation, and to ease the test requirements classification (step **Classify and document each of the test requirements**) are described in (Cagnin et al., 2004).

In step **Derive and document test cases**, test cases are derived from the defined requirements. One test requirement can generate more than one test case, for example, TC7 and TC8 test cases of Table 8. We suggest the tabular format for the test cases documentation. The documentation of each test case derived from the test requirements that are common to the majority of the patterns should contain the following information: **test case number**; **test requirement number** (obtained from the test requirement documentation); **operation** type (obtained from the test requirement documentation); **previous validations**, i.e., number of test cases that should be considered as pre-condition; **input data** (specifies the value that should be used as input data for the test); and **expected output** (specifies the test expected value relative to the input data). It can be noticed that some information of the test requirement documentation is repeated to ease the test case readability. In Table 8 a snippet of the test case documentation is presented, which was derived from the test requirements presented in Table 5 (that table presented those requirements that are common to most GRN patterns).

To document the test cases derived from the business test requirements, the following information should be added: **pattern name** (obtained from the test requirement name) and **class name** (obtained from the test requirement documentation). In Table 7, an example of the business test cases documentation for GRN pattern 4 (RENT THE RESOURCE) is presented. They were derived from the test requirements presented in Table 6.

In step **Map common test cases**, a mapping is done between the pattern specific attributes and the test cases that are common to all patterns. For that, it is necessary to know, for each pattern attribute, its type and possible length, whether or not this attribute is a primary key in the corresponding relational database table, and its relationships with other classes that participate of the pattern (as these are mapped to foreign keys in the relational database). Then, from the test requirements documentation, test cases derived from this requirement are obtained. For example, in the specific case of the `number` attribute (`Resource Rental` class of RENT THE RESOURCE pattern), which is an integer and primary key, we can reuse the test cases shown in Table 8, from the test requirements shown in Table 5.

Table 9 shows an example that illustrates the mapping between the test cases and the `number` attribute of the `Resource Rental` class (pattern 4 - RENT THE RESOURCE). In this specific case, two columns were added: **RDBMS table** and **attribute**, corresponding to the table and the attribute being validated.

In step **Make test resources available**, the documentation of the business test resources is placed in a new pattern section, named *VV&T Information*. For example, the equivalence classes presented in Table 4, the test requirements presented in Table 6, the test cases presented in Table 7 and the mapping presented in Table 9 are made available.

The documentation of the consistency and integrity test resources, which are applicable to all patterns, is made available at the end of GRN patterns, in a new section, also named *VV&T Information* (Tables 3, 5 and 8). In both sections, the functional criteria used to create the test resources (i.e., “equivalence partitioning” and “boundary value analysis”) should be mentioned.

Table 6: Partial Documentation of the business test requirements for the "Resource Rental" class of GRN pattern 4

Test Req Number	Type	Pattern	Pattern Class	Operation	RDB Table	Equiv Classes	Test Req Espec	Valid Conditions	Previous Validations	Comment	Expected Return
TR10	Business	Pattern 4: Rent the Resource	Resource Rental	Include, Modify	Resource Rental	51	resource instance situation == "available"	resource situation == "available"	all test requirements that are useful to make the consistency of the attributes of the Resouce Rental class	–	well succeeded verification, operation can proceed
TR11	Business	Pattern 4: Rent the Resource	Resource Rental	Include, Modify	Resource Rental	52	resource instance situation == "rented"	resource situation == "available"	all test requirements that are useful to make the consistency of the attributes of the Resouce Rental class	–	business functionality verification error, operation cannot proceed
…	…	…	…	…	…	…	…	…	…	…	…

Table 7: Business Test Cases for the "Rent the Resource" GRN pattern

Test Case Number	Test Req Number	Operation	Pattern	Pattern Class	Previous Validations	Input Data	Expected Output
TC12	TR10	Include, Modify	Pattern 4: Rent the Resource	Resource Rental	all test cases that are useful to ensure the consistency of the attributes of the Resource Rental class	situation := "1" (this means, to the GREN framework, that the resource is available)	well succeeded business functionality verification, operation can proceed
TC13	TR11	Include, Modify	Pattern 4: Rent the Resource	Resource Rental	all test cases that are useful to make the consistency of the attributes of the Resouce Rental class	situation := "2" (this means, to the GREN framework, that the resource is rented)	business functionality verification error, operation cannot proceed
…	…	…	…	…	…	…	…

Table 8: Consistency and Integrity test cases for GRN patterns

Test case number	Test Req number	Operation	Previous Validations	Input Data	Expected Output
TC01	TR01	Include, Modify, Delete, Search	–	attribute := "A"	consistency verification error, operation cannot proceed
TC02	TR02	Include, Modify, Delete, Search	–	attribute := 0	consistency verification error, operation cannot proceed
TC3	TR03	Include, Modify, Delete, Search	–	attribute := 1	well succeeded verification, operation can proceed
TC4	TR04	Include, Modify, Delete, Search	–	attribute := 2147483647	well succeeded verification, operation can proceed
TC5	TR04	Include, Modify, Delete, Search	–	attribute := 2147483646	well succeeded verification, operation can proceed
TC6	TR05	Include, Modify, Delete, Search	–	attribute := 2147483648	verification error, operation cannot proceed
TC7	TR06	Include, Modify, Delete, Search	–	attribute := -1	verification error, operation cannot proceed
TC8	TR06	Include, Modify, Delete, Search	–	attribute := 0	verification error, operation cannot proceed
TC9	TR07	Include, Modify, Delete, Search	–	attribute := null	verification error, operation cannot proceed
TC10	TR08	Include	TC01 to TC09	attribute := 1 (already registered)	integrity verification error, operation cannot proceed
TC11	TR09	Modify, Delete, Search	TC01 to TC09	attribute := 2 (not registered)	integrity verification error, operation cannot proceed
...	...	...	...	...	...

Table 9: Test Cases Mapping

Pattern	Pattern Class	RDBMS Table	Attribute	Test Case number	Previous Validations
Pattern 4: Rent the Resource	Resource Rental	ResourceRental	number	TC1 to TC11	–
Pattern 4: Rent the Resource	Resource Rental	ResourceRental	observation	...	TC1 to TC9
...	...	...	...	...	...

To reuse the test resources, the reuse guidelines proposed in (Cagnin et al., 2004) should be used. Basically, they consist of mapping the system functionality to the associated GRN patterns and reusing the test resources available, both the specific ones and those that are common to several patterns. For system functionality that doesn't correspond to GRN patterns, it is also possible to reuse, maybe after some adaptation, the test resources that are common to all patterns. Furthermore, guidelines are provided to test the correct use of the pattern language as a whole, as several issues need to be checked, such as: whether the pattern sequence was used correctly, according to the GRN structure (Figure 1) and the *following patterns* section; whether the mandatory classes of the pattern were considered, according to the pattern sections *structure*, *participants* and *variants*; and, when the usage of one pattern requires the application of other patterns, if they were correctly used. We suggest that all these guidelines are applied incrementally for each system functional requirement.

5. Case Study

Two reengineering case studies were conducted with the support of the PARFAIT process (Cagnin et al., 2003b). This process uses the GREN framework for computational support. As GREN was built based on the GRN pattern language, all classes and relationships contained in each GRN pattern have a corresponding implementation within the GREN classes. Variants of each pattern were implemented as framework *hot spots* [5].

[5] *hot spots* are framework abstract classes or methods that must be overridden in the specific application during framework instantiation (Markiewicz and Lucena, 2001).

As GRN belongs to the same domain of the legacy system, it is used to support its understanding and to build its class diagram. Patterns and variants applied in the construction of the class diagram are used in the GREN framework instantiation. Users of the legacy system should participate in the reengineering to refine and validate the resulting artifacts.

In PARFAIT, the legacy system understanding and the identification of the business rules are also obtained by executing the system functions, which is done in a systematic way with the support of VV&T activities. Test cases executed in the legacy system are used later to validate the target system.

The legacy system submitted to reengineering in the case studies is a small system (with approximately 6 KLOC) to control book loans in a University library. It was developed in Clipper and was operative during the case studies conduction.

The first case study was done by an undergraduate student in the fourth year of a Computing Science course at ICMC-USP. The test cases used to execute the legacy system and, afterwards, to validate the target system, were created from scratch with the support of functional test criteria, namely the "equivalence partitioning" and the "boundary value analysis", totalizing 174 equivalence classes and 354 test cases. This was done in about 549:00 hours from a total of 676:29 hours spent on the reengineering, i.e., approximately 81% of the reengineering effort was spent with VV&T activities. More information about this case study can be found elsewhere (Cagnin et al., 2003a).

The second case study was conducted by an undergraduate student of the second semester of a Computing Science course at ICMC-USP. In this case, the test cases used to execute the legacy system and, afterwards, to validate the target system, were reused, whenever possible, from the *VV&T Information* sections of the GRN patterns used to model the legacy system. The creation of most test cases was totally reused, as presented in Table 10. This consumed 238:10 hours from a total of 323:40 hours spent in the case study, i.e., approximately 74% of the reengineering effort was spent to create test cases. The time spent with VV&T activities (i.e., time spent with test creation and execution) had a reduction of about 57%, compared with the first case study, and the percentage in relation to the total reengineering time suffered meaningful reduction of about 52%. More information about the conduction of this case study can be found in (Cagnin, 2005).

Table 10: Data about VV&T activities collect in the second reengineering case study

Data collected	Value	Percentage
Test cases created from reused equivalence classes	34	4%
Test cases created from scratch (new equivalence classes)	43	5%
Test cases created from reused and adapted requirements	80	9,4%
Test cases created from reused test requirements	695	81,6%

The results indicate that there is a reduction in the time spent with tests both in reengineering and in software development when test resources are allocated to patterns to ease their reuse. However, we should observe that these results are only clues that the reengineering time is reduced. Controlled experiments should be conducted to verify this hypothesis.

6. Conclusion and Future Work

As mentioned in Section 2, a present concern exists in providing test resources associated to different reuse paradigms (Tsai et al., 1999; Weyuker, 1998; Mariani et al., 2004; Tevanlinna et al., 2004), to guarantee the quality and reliability of the products created. Nevertheless, the authors of this paper did not see any evidence in the literature of work to capture expert solutions as well as the underlying validation aspects. Another problem we see is that we frequently cannot assess the quality of a pattern we want to use, because it does not provide indications of how it was validated, except by the known uses section, which is often vague. This paper presented a strategy to solve these problems, by including a section in the pattern documentation to help validate the solution in the pattern. Existing pattern formats, as for example Gamma et al. (1995), Appleton (1997) and Meszaros and Doble (1998), do not consider this aspect.

The proposed strategy was used in some patterns of the GRN pattern language that belongs to the business resource management domain. The test resources were reused during a reengineering case study and the time spent was compared with another case study performed, without the reuse of test resources. The results have suggested a reduction of the time spent with VV&T activities. However, controlled case studies need to be performed to verify these results.

The proposed strategy was used with only a few GRN patterns, which are analysis patterns. So, it would be desirable to apply the strategy to other patterns, with different contexts and in other domains, using other test criteria, so that the strategy can be refined and generalized. We intend to explore these ideas next in the context of Web applications development.

References

Appleton, B. (1997). Patterns and software: Essential concepts and terminology. site. `http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html`. Accessed: December, 2003.

Beck, K. (2002). *Test-driven development: by example*. The Addison-Wesley signature series. Addison-Wesley, first edition.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development. site. `http://www.agilemanifesto.org`. Accessed: June, 2003.

Beck, K. and Johnson, R. (1994). Patterns generate architectures. In *ECOOP'1994, 8th European Conference on Object-Oriented Programming*, pages 139–149, Bologna, Italy.

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, first edition.

Braga, R. T. V., Germano, F. S. R., and Masiero, P. C. (1999). A pattern language for business resource management. In *PLOP'1999, 6th Conference on Pattern Languages of Programs*, pages 1–33, Urbana, IL, USA.

Braga, R. T. V. and Masiero, P. C. (2002). A process for framework construction based on a pattern language. In *COMPSAC'2002, 26th Annual International Computer Software and Applications Conference, 26*, pages 615–620, Oxford, England. IEEE.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., and Stal, M. (1996). *Pattern-oriented software architecture: A System of Patterns*. Wiley Series in Software Design Patterns. Wiley, first edition.

Cagnin, M. I. (2005). *PARFAIT: A Contribution for Software Reengineering based on Pattern Languagens and Frameworks*. PhD thesis, Instituto de Ciências Matemáticas e de Computação, Universidade of São Paulo, São Carlos–SP. (in Portuguese).

Cagnin, M. I., Maldonado, J. C., Chan, A., Penteado, R. D., and Germano, F. S. (2004). Reuse on Testing Activity to Reduce Cost and Effort of VV&T in Software Development and Reengineering. In *XVIII Brazilian Software Engineering Symposium*, pages 71–85, Brasília-DF, Brazil. (in Portuguese).

Cagnin, M. I., Maldonado, J. C., Germano, F. S., Chan, A., and Penteado, R. D. (2003a). A reengineering case study using PARFAIT process. In *SDMS'2003, Naive Software Development and Maintenance Symposium*, pages 1–10, Rio de Janeiro, RJ. (in Portuguese).

Cagnin, M. I., Maldonado, J. C., Germano, F. S., and Penteado, R. D. (2003b). PARFAIT: Towards a framework-based agile reengineering process. In *ADC'2003, Agile Development Conference*, pages 22–31, Salt Lake City, UTHA, USA. IEEE.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley.

Coad, P. (1992). Object-oriented patterns. *Communications of the ACM*, 35(9):152–159.

Coad, P., North, D., and Mayfield, M. (1997). *Object models: Strategies, patterns and applications*. Yourdon Press, second edition.

DeLano, D. E. and Rising, L. (1998). *Pattern Languages of Program Design 3*, volume 1, chapter Software Design Patterns: Common Questions and Answers, pages 503–525. Addison-Wesley, first edition.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2000). A pattern language for reverse engineering. In *EuroPLOP'2000, 5th European Conference on Pattern Languages of Programming and Computing*, pages 189–208, Irsee, Germany. Andreas Ruping.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns Elements of Reusable of Object-Oriented Software*. Addison-Wesley, second edition.

Germany Ministry of Defense (1992). V-Model: Software Lifecycle Process Model. Technical Report General Reprint Nº250, Germany Ministry of Defense.

Harrold, M. J. (2000). Testing: a roadmap. In *ICSE'2000, 22nd International Conference on Software Engineering, The Future of Software Engineering*, pages 61–72, New York, NY, USA. ACM Press.

Hayes, J. H. and Offutt, A. J. (1999). Increased software reliability through input validation analysis and testing. In *ISSRE'1999, 10th International Symposium on Software Reliability Engineering*, pages 199–209, Boca Raton, FL, USA.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, third edition.

Lemos, G. S. (2002). PRE/OO - an object-oriented reengineering process emphasing quality assurance. M.sc. dissertation, Computer Department. Federal University of São Carlos, São Carlos-SP, Brazil. 159 p. (in Portuguese).

Mariani, L., Pezzè, M., and Willmor, D. (2004). Generation of selft-test components. *Lecture Notes in Computer Science (LNCS), Springer*, 3236(2004):337–350.

Markiewicz, M. and Lucena, C. (2001). Object oriented framework development. *ACM Crossroads Student Magazine*. Crossroads 7.4, Summer 2001. `http://acm.org/crossroads/xrds7-4/frameworks.html`. Accessed: January/2005.

McGregor, J. D. (2001). Structuring test assets in a product line effort. In *Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications*, pages 89–92.

Meszaros, G. and Doble, J. (1998). *A pattern language for pattern writing*, chapter 29, J. Coplien; D. Schmidt. Pattern Languages of Program Design, pages 529–574. Reading-MA, Addison-Wesley.

Myers, G. J. (2004). *The art of software testing*. Wiley, second edition.

MySQL (2003). MySQL Reference Manual. `http://www.mysql.com/doc/en/index.html`. Accessed: December/2003.

Pressman, R. (2001). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 5th edition.

Recchia, E. (2002). Reverse engineering and reengineering based on patterns. Master's thesis, Computer Department. Federal University of São Carlos, São Carlos-SP, Brazil. 159 p. (in Portuguese).

Rocha, A. R., Maldonado, J., and Weber, K. (2001). *Software Quality: Teory and Practice*. Prentice Hall, first edition. (in Portuguese).

Roper, M. (1994). *Software Testing*. The International Software Engineering Series. McGraw-Hill.

Sommerville, I. (2000). *Software Engineering*. Addison-Wesley, sixth edition.

Stevens, P. and Pooley, R. (1998). Systems reengineering patterns. In *ACM SIGSOFT 1998, Sixth Internatinal Symposium on the Foundations of Software Engineering*, pages 17–23, Orlando, Florida, USA.

Tevanlinna, A., Taina, J., and Kauppinen, R. (2004). Product family testing: a survey. *ACM SIGSOFT Software Engineering Notes*, 29(2):12–12.

Tsai, W., Tu, Y., Shao, W., and Ebner, E. (1999). Testing extensible design patterns in object-oriented frameworks through scenario templates. In *COMPSAC'1999, 23rd International Computer Software and Applications Conference*, pages 166–171, Phoenix, AZ.

Weyuker, E. J. (1998). Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59.

Acknowledgements

The authors would like to thank Linda Rising, who provided several comments to improve this paper.

XSpeed: Uma ferramenta para geração de aplicações distribuídas baseadas em padrões

Lincoln S. Rocha[1], Rute Nogueira[2], João Gustavo Prudêncio, Rossana M. C. Andrade e Jerffeson Teixeira de Souza

Universidade Federal do Ceará, Departamento de Computação, Campus do Pici
Bloco 910, 60455-760, Fortaleza – Ceará – Brasil.

{lincoln,rute,gustavo,rossana,jeff}@lia.ufc.br

Resumo. *Este artigo apresenta uma ferramenta, denominada XSpeed, que tem como objetivo principal aumentar a produtividade no desenvolvimento de aplicações para ambiente distribuído utilizando padrões para resolver problemas recorrentes neste domínio. XSpeed recebe como entrada um arquivo XMI contendo o modelo UML de uma aplicação e então realiza a geração automática de código para uma plataforma específica. Também neste artigo é apresentado um estudo de caso para ilustrar o funcionamento da ferramenta.*

Abstract. *This paper presents a tool, called XSpeed, which increases productivity in the development of applications for a distributed environment using software patterns to solve recurrent problems in the scope of this domain. XSpeed receives as input a XMI file, which has the UML model of an application and then performs an automatic code generation for a specific platform. A case study is presented to illustrate how this tool works to achieve its requirements.*

1. Introdução

O desenvolvimento de aplicações para o ambiente distribuído é um processo complexo e que requer do engenheiro de *software* um maior esforço. No intuito de padronizar e simplificar esta tarefa, especificações como CORBA [14] e J2EE [21] surgem como uma alternativa prática e viável. Da mesma forma, a utilização de padrões de *software* e *frameworks*, neste contexto, podem contribuir de maneira significativa para o aumento da reusabilidade, escalabilidade, flexibilidade [18] e diminuição da complexidade do código produzido. Em contrapartida, a incorporação de especificações, padrões de *software* e *frameworks* ao processo de desenvolvimento pode ocasionar um impacto, não positivo, na curva do aprendizado.

A utilização de ferramentas de geração automática de aplicações é uma abordagem que traz consigo inúmeras vantagens. Além de aumentar a produtividade na fase de desenvolvimento, garante a não ocorrência de erros na tradução da especificação (modelo de alto nível) para a implementação (linguagem alvo), uma vez que essa atividade é realizada de maneira automática. Além disso a manutenibilidade da aplicação gerada é facilitada devido à uniformidade e modularidade do código produzido.

---

[1] Bolsista de mestrado financiado pela CAPES.
[2] Bolsista de mestrado financiado pela FUNCAP.

Este artigo propõe uma ferramenta denominada XSpeed, que busca integrar tecnologias e facilitar a reutilização de padrões no processo de desenvolvimento de aplicações para ambiente distribuído através da geração automática de código a partir de modelos UML [7], disponibilizados através de arquivos no formato XMI (*XML Metadata Interchange*) [15]. Desta maneira, as novas aplicações geradas seguirão um modelo de arquitetura padronizado, facilitando o entendimento e a manutenibilidade do sistema como um todo.

O restante deste artigo é descrito como segue: na seção 2 encontram-se padrões de *software* e ferramentas relacionadas com o desenvolvimento de aplicações para um ambiente distribuído; na seção 3 são apresentados os padrões utilizados para fazer a validação da ferramenta; na seção 4 é apresentada uma descrição da arquitetura do XSpeed; na seção 5 é mostrado um estudo de caso da ferramenta e por fim na seção 6 são apresentadas as conclusões obtidas neste trabalho bem como sugestões para trabalhos futuros.

2. Trabalhos Relacionados

A reutilização de padrões no desenvolvimento de um *software* tem emergido como uma das mais promissoras abordagens para a melhoria da qualidade dos artefatos de *software*, pois eles permitem que a experiência de desenvolvedores seja documentada e reutilizada, registrando-se soluções de projeto para um determinado problema em um contexto particular.

A utilização de ferramentas que fazem geração automática de aplicações e utilizam padrões de *software* como estratégia para a geração de código tem se tornado uma alternativa viável para diminuir tempo e esforço gastos no processo de desenvolvimento e manutenção de *softwares*. Nesta seção são apresentados padrões e ferramentas que buscam solucionar problemas envolvidos com o desenvolvimento de aplicações distribuídas.

Em [8], os autores apresentam os padrões DAP-EJB (Distributed Adapters Pattern with EJB) e PDC-EJB (Persistent Data Collections with EJB) que auxiliam na estruturação de aplicações EJB [23]. Essa estruturação pode acontecer em sistemas já existentes, sem EJB, bem como em projetos de novos sistemas, os quais obterão alguns benefícios como reusabilidade, extensibilidade, modularidade, independência de tecnologia (distribuição ou dados) e desempenho.

Além disso, em [17] tem-se o Padrão de DBCD (Desenvolvimento Baseado em Componentes Distribuídos), cuja intenção é a criação de componentes distribuídos reutilizáveis para diferentes domínios de aplicações. Este padrão trabalha com a integração de diferentes tecnologias em uma ferramenta CASE, para apoiar o Desenvolvimento Baseado em Componentes (DBC), além de cobrir todo o ciclo de vida dos componentes distribuídos e definir mudanças no código de comunicação dos mesmos.

Em [3] é apresentado um método de implementação que orienta a transformação progressiva que torna uma aplicação inicialmente centralizada em uma distribuída. O método ameniza a complexidade inerente a sistemas distribuídos e torna os testes mais efetivos. Além disso, esse método utiliza o Distributed Adapters Pattern (DAP) apresentado em [2] promovendo uma maior modularidade, reuso, extensibilidade assim como uma implementação progressiva.

A ferramenta Cordel apresentada em [10] promove de forma flexível a geração, compilação e implantação automática de sistemas Web, com base em tecnologias de

objetos distribuídos, a partir de modelos UML, seguindo especificações e tecnologias já consolidadas. Esses sistemas que adotam mecanismos que favorecem a construção de aplicações reutilizáveis, flexíveis e escaláveis trazem consigo a conseqüência natural de um maior esforço no processo de desenvolvimento, devido à adoção de especificações, como J2EE ou CORBA que, por sua vez, implicam numa maior necessidade de aprendizado da tecnologia e uma maior quantidade de código a ser escrito.

Além da ferramenta Cordel, pode-se destacar como importante para este trabalho a ferramenta AndroMDA [4], que trabalha com a geração de código fonte a partir da especificação UML. Como conseqüência, o código fonte torna-se pouco relevante, pois os diagramas UML é que são essenciais para a implementação. Além disso, a ferramenta sugere o uso de diversas tecnologias para persistência dos dados como *Hibernate, Spring* e *SOAP*, ao mesmo tempo em que se propõe a diminuir o tempo de desenvolvimento de programas e promover o uso intensivo de padrões.

3. Padrões Utilizados

Alguns padrões foram selecionados e estudados com a finalidade de fazer a validação da ferramenta proposta neste artigo. O critério de seleção levou em consideração a adequação do padrão à solução proposta. Uma seleção mais rigorosa foi deixada para um momento posterior. A seguir, os padrões escolhidos são apresentados.

O padrão BD (*Business Delegate*) [1], normalmente, é utilizado para reduzir o acoplamento entre os clientes da camada de apresentação e os serviços de negócios. O BD oculta os detalhes de implementação por trás do serviço de negócios, como forma de pesquisa e acesso em ambientes remotos. A **Figura 1** mostra o diagrama de classes do padrão BD.

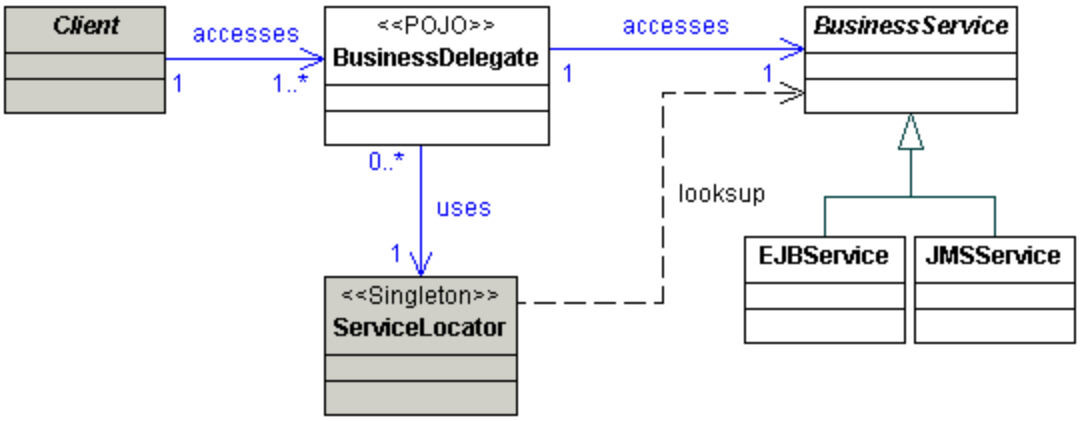


Figura 1. Diagrama de classes do BD [1]

Já o BO (*Business Object*) [1] é empregado para separar dados e lógica de negócio usando um modelo orientado a objeto. O BO fará, portando, o encapsulamento das regras de negócio e do gerenciamento da camada de persistência. Na **Figura 2** é exposto o diagrama de classes do BO.

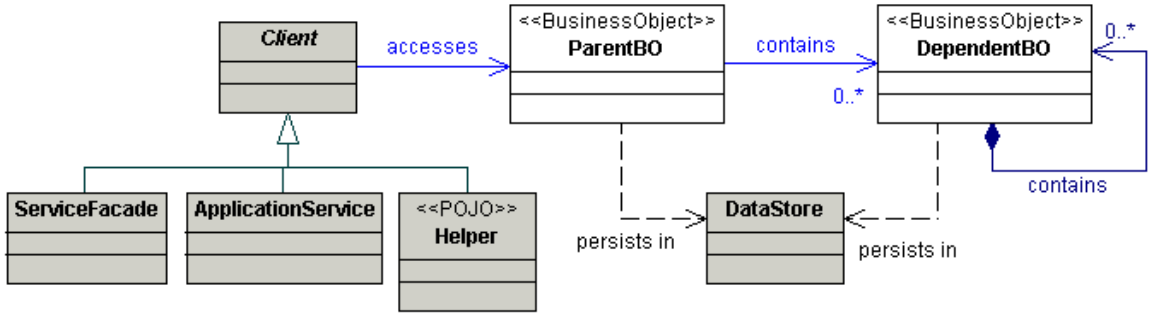


Figura 2. Diagrama de classes do BO [1]

Segundo [1], o acesso a dados muda dependendo da sua origem. O acesso ao armazenamento persistente, como em um banco de dados, varia muito dependendo do tipo de armazenamento e da implementação do desenvolvedor. Através da utilização do padrão DAO (*Data Access Object*) pode-se extrair e encapsular todos os acessos à origem de dados em um único objeto. O DAO gerencia a conexão com a fonte de dados para obter e armazenar dados. Seu diagrama de classes pode ser observado na **Figura 3**.

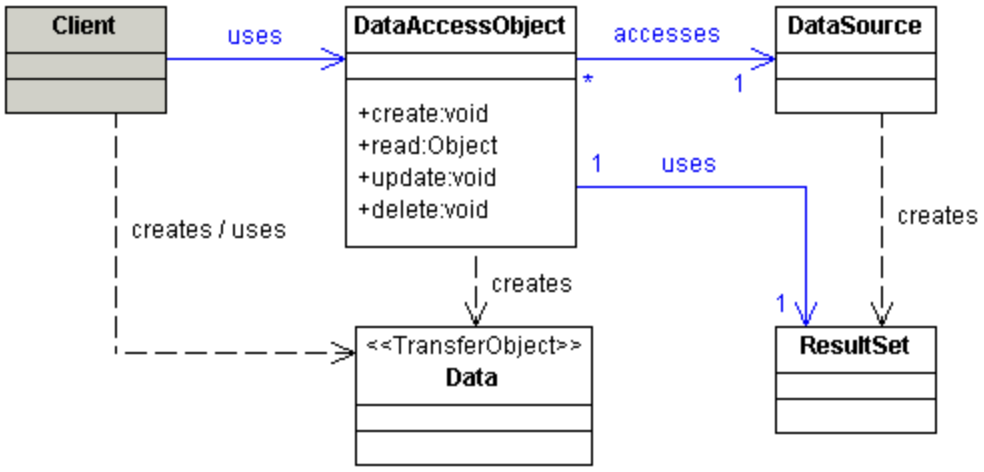


Figura 3. Diagrama de classes do DAO [1]

4. XSpeed e sua Arquitetura

XSpeed é uma ferramenta que tem como objetivo principal aumentar a produtividade no desenvolvimento de aplicações para ambiente distribuído utilizando padrões para resolução de problemas recorrentes. A ferramenta recebe como entrada um arquivo no formato XMI contendo o modelo UML da aplicação a ser gerada. Em seguida, mapeia os padrões que deverão ser aplicados como estratégia para resolução de problemas e finalmente faz a geração automática do código interligando as tecnologias específicas para a resolução de cada um dos problemas inerentes ao domínio.

A arquitetura do XSpeed, demonstrada na **Figura 4**, se divide em três módulos básicos: módulo de conversão, módulo de configuração e módulo de geração.

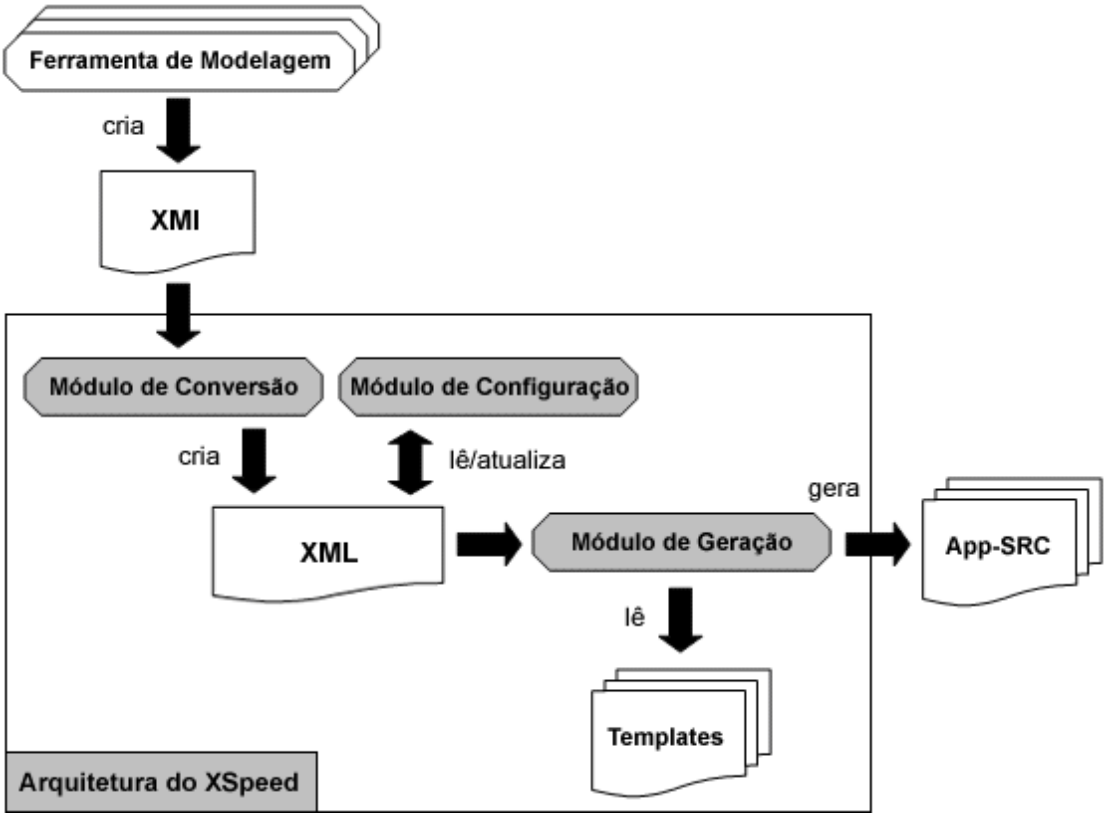


Figura 4. Arquitetura geral da ferramenta XSpeed

4.1. Módulo de Conversão

O módulo de conversão faz a extração das informações relacionadas a cada uma das classes do modelo UML contido no arquivo XMI, disponibilizando-as como uma instância do modelo de representação descrito na **Figura 5**. Em seguida, esta instância é armazenada em um arquivo XML intermediário que servirá como base para geração da aplicação.

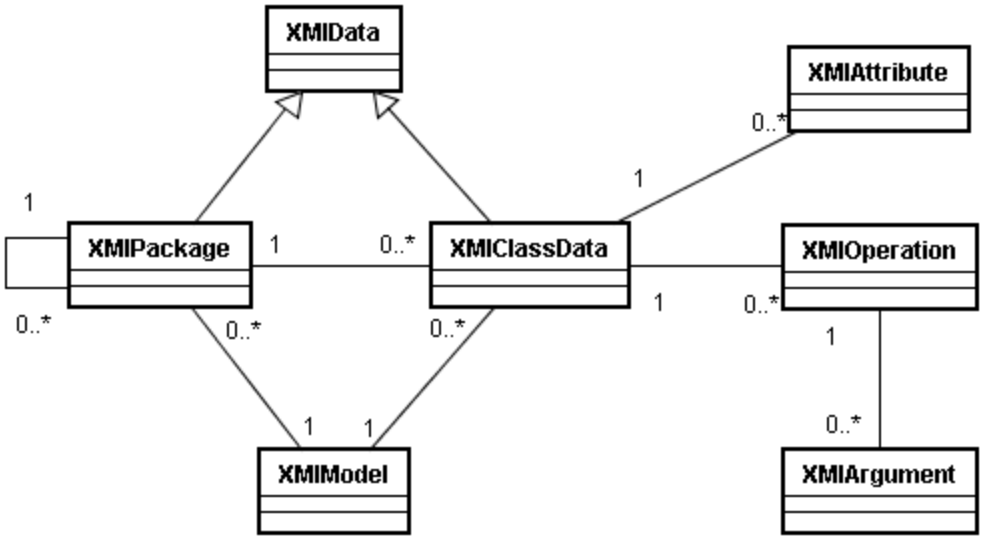


Figura 5. Modelo de mapeamento do XMI

Nesta fase, uma filtragem por meio de um parser é concebida sobre o arquivo XMI de entrada. Apenas informações relevantes para o XSpeed são selecionadas, tais como a estrutura e o relacionamento das classes do modelo. Desse modo, não são avaliadas as informações contidas em outros possíveis diagramas como os de casos de uso, colaboração e seqüência.

4.2. Módulo de Configuração

Neste módulo, é feita uma seleção manual dos padrões e das tecnologias, de persistência e acesso remoto, a serem aplicadas sobre as classes do modelo para resolução dos problemas específicos do domínio. XSpeed propõe um conjunto de padrões para resolução de problemas específicos de aplicações distribuídas tais como delegação de serviço, processamento de lógica de negócio e persistência de dados. A versão atual da ferramenta não disponibiliza uma interface gráfica para realizar os mapeamentos. O usuário pode utilizar algum editor de XML para alterar o mapeamento *default* do arquivo XML intermediário gerado pela ferramenta.

Para resolver o problema de acesso remoto à lógica de negócio, adotou-se o padrão BD, apresentado na seção 3, integrado ao *framework Spring* [19] [26] que fornece uma estrutura transparente de comunicação distribuída baseada em RMI (*Remote Method Invocation*) [24]. De acordo com o diagrama de classes observado na **Figura 6**, o código cliente faz apenas chamadas locais. Assim sendo, toda a complexidade das chamadas remotas são encapsuladas pelo BD. Além de facilitar o desenvolvimento da aplicação cliente, o seu simples uso promove a separação explícita entre a camada de apresentação e a tecnologia envolvida na camada de negócio.

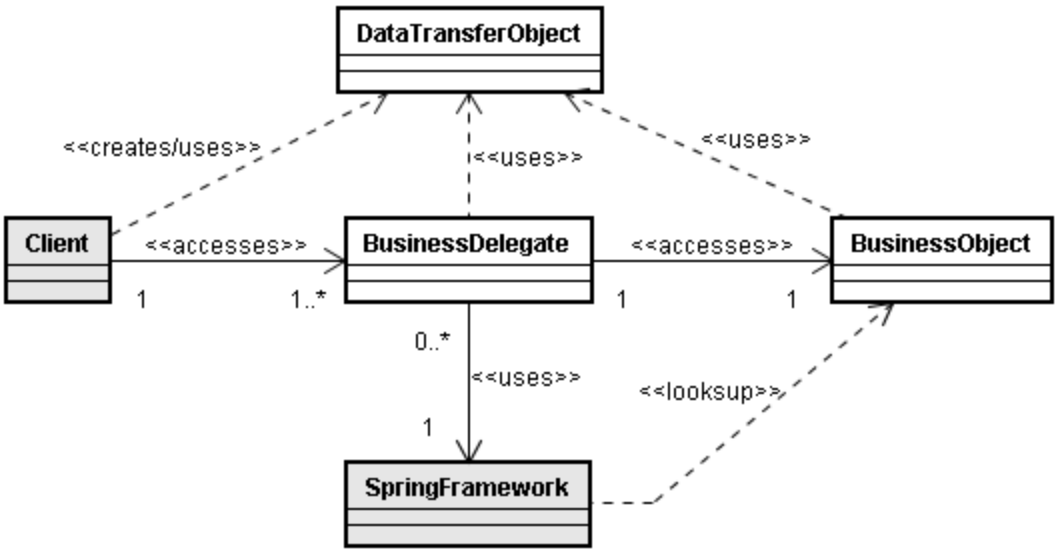


Figura 6. Diagrama de classes do BD adaptado

Na camada de lógica de negócio adotou-se o padrão BO, descrito na seção 3, que faz o encapsulamento das regras de negócio e do gerenciamento da camada de persistência. O acesso à camada de persistência pode acontecer de maneira remota (**Figura 7**), portanto, o BO também faz uso do *framework Spring*.

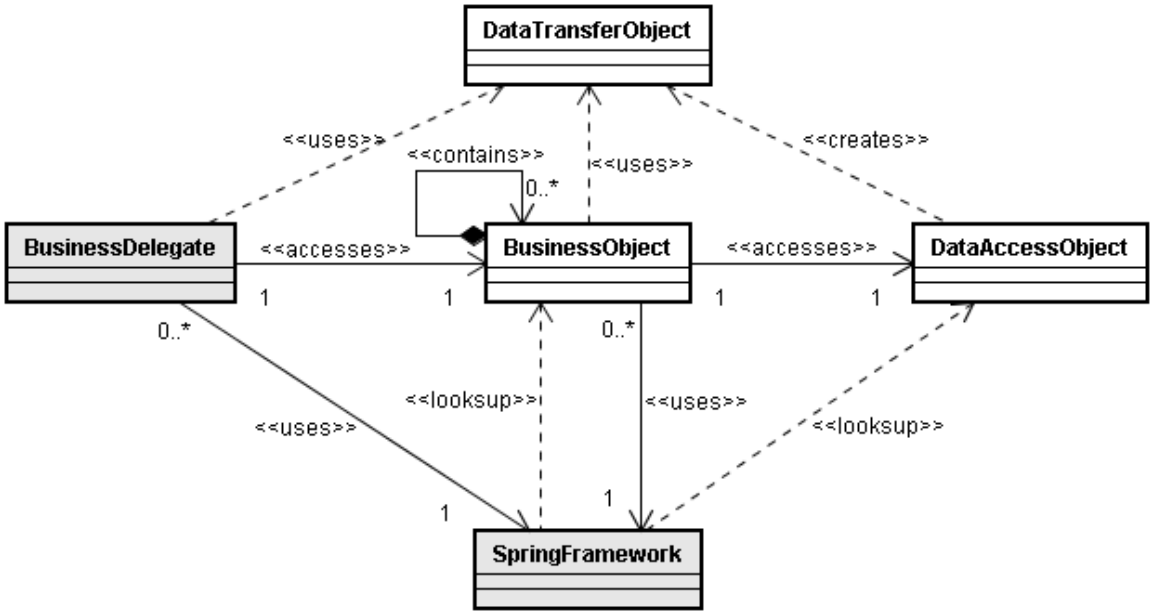


Figura 7. Diagrama de classes do BO adaptado

Por fim, elegeu-se o padrão DAO, explanado na seção 3, como técnica para encapsular as diferentes estratégias de persistência da aplicação. Atualmente XSpeed possibilita a integração de três tecnologias de persistência remota JDBC [22] [16] nativo, *Hibernate* [11] [6] e JPox [13] (**Figura 8**) sendo que as duas últimas fazem uso da primeira para persistir dados de maneira transparente.

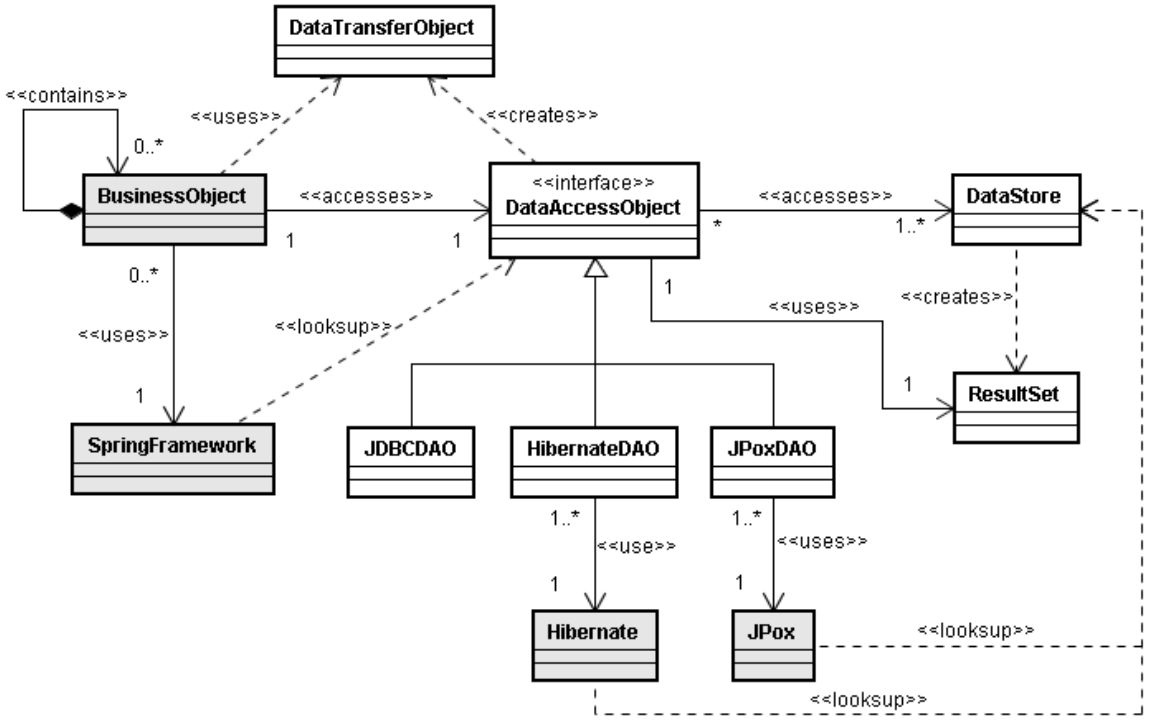


Figura 8. Diagrama de classes do DAO adaptado

4.3. Módulo de Geração

Este módulo é responsável por fazer a geração de código de todas as classes da aplicação. XSpeed faz uma junção entre as informações contidas no arquivo XML intermediário e os *templates* (**Figura 9**) que definem a estrutura das classes, inclusive o formato dos padrões a serem aplicados. Só então a geração de código é realizada para uma plataforma específica. A versão atual da ferramenta possibilita a geração apenas para a plataforma Java [20].

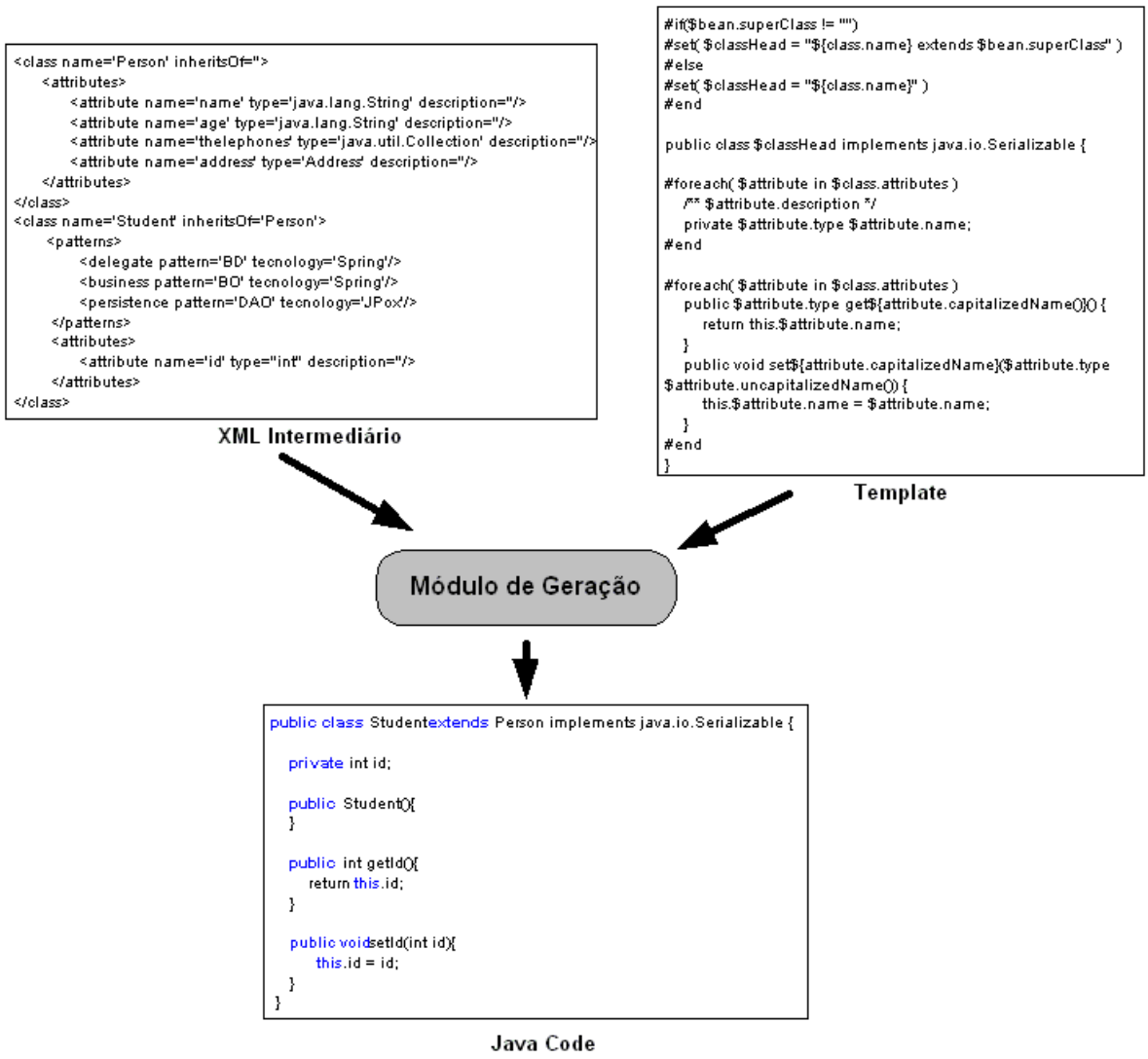


Figura 9. Processo de geração de código

O motor de geração do XSpeed utiliza o *Velocity Template Engine* [5] para fazer o *merge* entre as informações extraídas do arquivo XML intermediário e os *templates*. O *Velocity* possui uma linguagem de manipulação de *template*: a VTL (*Velocity Template Language*), que permite fazer a inserção de informações de maneira dinâmica dentro do template. O trecho de código (**Figura 10**) a seguir mostra a utilização da VTL para fazer a geração dos atributos de uma classe e seus respectivos métodos de acesso.

```
#foreach( $attribute in $bean.attributes )
    /** $attribute.description */
    private $attribute.type $attribute.name;
#end

#foreach( $attribute in $bean.attributes )
    public $attribute.type get${attribute.capitalizedName()}() {
        return this.$attribute.name;
    }
    public void set${attribute.capitalizedName}($attribute.type $attribute.uncapitalizedName()) {
        this.$attribute.name = $attribute.name;
    }
#end
```

Figura 10. Template Velocity

5. Estudo de Caso

Nesta seção são mostradas todas as etapas para geração de uma aplicação para ambiente distribuído utilizando XSpeed. A aplicação escolhida como exemplo consiste na manutenção de um cadastro simplificado de estudantes universitários. O diagrama de classes da **Figura 11** representa o modelo UML da aplicação.

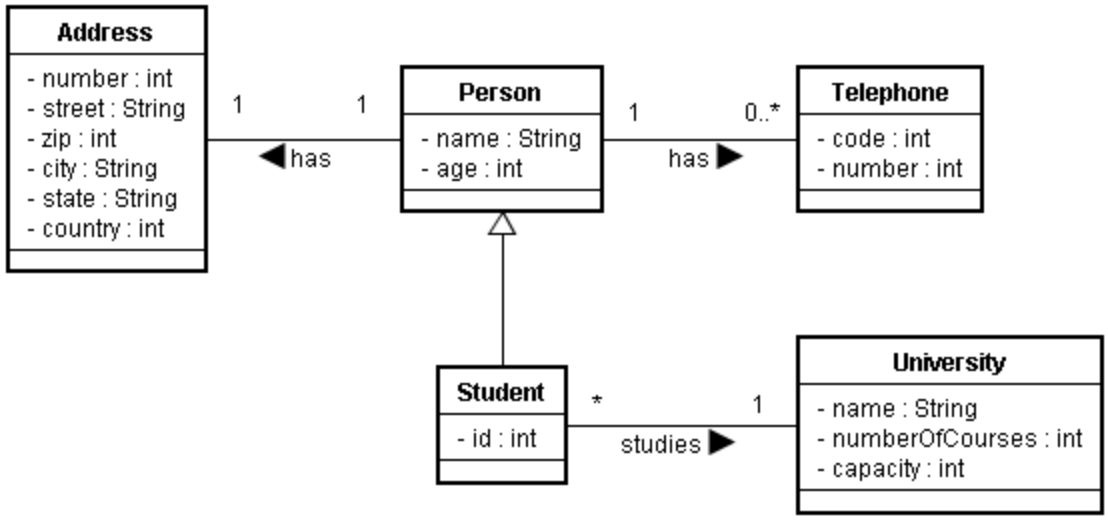


Figura 11. Diagrama de classes da aplicação

O modelo exibido na **Figura 11** pode ser criado e convertido para o formato XMI por meio de uma ferramenta de modelagem UML, tais como Rational Rose [12], ArgoUML [25] e Poseidon [9]. A **Figura 12** mostra a descrição das classes *Person* e *Student* no formato resultante da conversão.

```
<UML:Class xmi.id = 'I1b8378fm103281c4dc2mm7f3e' name = 'Person' visibility = 'public'
  isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
  isActive = 'false'>
  <UML:Classifier.feature>
    <UML:Attribute xmi.id = 'I1b8378fm103281c4dc2mm7ecc' name = 'name' visibility = 'private'
      isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
      <UML:StructuralFeature.type>
        <UML:Class xmi.idref = 'I1b8378fm103281c4dc2mm7ebb'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
    <UML:Attribute xmi.id = 'I1b8378fm103281c4dc2mm7eba' name = 'age' visibility = 'private'
      isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
      <UML:StructuralFeature.type>
        <UML:DataType xmi.idref = 'I1b8378fm103281c4dc2mm7ef2'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
  </UML:Classifier.feature>
</UML:Class>
<UML:Class xmi.id = 'I1b8378fm103281c4dc2mm7f18' name = 'Student' visibility = 'public'
  isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
  isActive = 'false'>
  <UML:GeneralizableElement.generalization>
    <UML:Generalization xmi.idref = 'I1b8378fm103281c4dc2mm7d7c'/>
  </UML:GeneralizableElement.generalization>
  <UML:Classifier.feature>
    <UML:Attribute xmi.id = 'I1b8378fm103281c4dc2mm7ea9' name = 'id' visibility = 'private'
      isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
      <UML:StructuralFeature.type>
        <UML:DataType xmi.idref = 'I1b8378fm103281c4dc2mm7ef2'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
  </UML:Classifier.feature>
</UML:Class>
```

Figura 12. Estrutura do arquivo XMI

Na fase de conversão, o arquivo XMI gerado é fornecido como entrada para XSpeed que, por sua vez, faz a filtragem das informações de cada uma das classes para o seu modelo de representação, descrito na seção 4.1. Em seguida, atualiza o arquivo XML intermediário incorporando as novas características da aplicação. A **Figura 13** mostra as classes *Person* e *Student* descritas no formato deste arquivo com o mapeamento *default* da ferramenta.

```
<class name='Person' inheristOf=''>
    <patterns>
        <delegate pattern='BD' tecnology='Spring'/>
        <business pattern='BO' tecnology='Spring'/>
        <persitence pattern='DAO' tecnology='JPox'/>
    </patterns>
    <attributes>
        <attribute name='name' type='java.lang.String' description=''/>
        <attribute name='age' type='java.lang.String' description=''/>
        <attribute name='thelephones' type='java.util.Collection' description=''/>
        <attribute name='address' type='Address' description=''/>
    </attributes>
</class>
<class name='Student' inheristOf='Person'>
    <patterns>
        <delegate pattern='BD' tecnology='Spring'/>
        <business pattern='BO' tecnology='Spring'/>
        <persitence pattern='DAO' tecnology='JPox'/>
    </patterns>
    <attributes>
        <attribute name='id' type='int' description=''/>
    </attributes>
</class>
```

Figura 13. Arquivo XML intermediário

No exemplo específico (**Figura 14**) foi removido, manualmente, o mapeamento dos padrões e das tecnologias que incidiam sobre a classe *Person*.

```
<class name='Person' inheristOf=''>
    <attributes>
        <attribute name='name' type='java.lang.String' description=''/>
        <attribute name='age' type='java.lang.String' description=''/>
        <attribute name='thelephones' type='java.util.Collection' description=''/>
        <attribute name='address' type='Address' description=''/>
    </attributes>
</class>
<class name='Student' inheristOf='Person'>
    <patterns>
        <delegate pattern='BD' tecnology='Spring'/>
        <business pattern='BO' tecnology='Spring'/>
        <persitence pattern='DAO' tecnology='JPox'/>
    </patterns>
    <attributes>
        <attribute name='id' type='int' description=''/>
    </attributes>
</class>
```

Figura 14. Arquivo XML intermediário mapeado

Após as alterações realizadas sobre o arquivo XML intermediário, o código é gerado. O diagrama de classes da **Figura 15** mostra o novo formato da aplicação com a incorporação das classes *StudentBDImpl*, *StudentBOImpl* e *StudentJPoxDAO* e das interfaces *StudentBO* e *StudentDAO*.

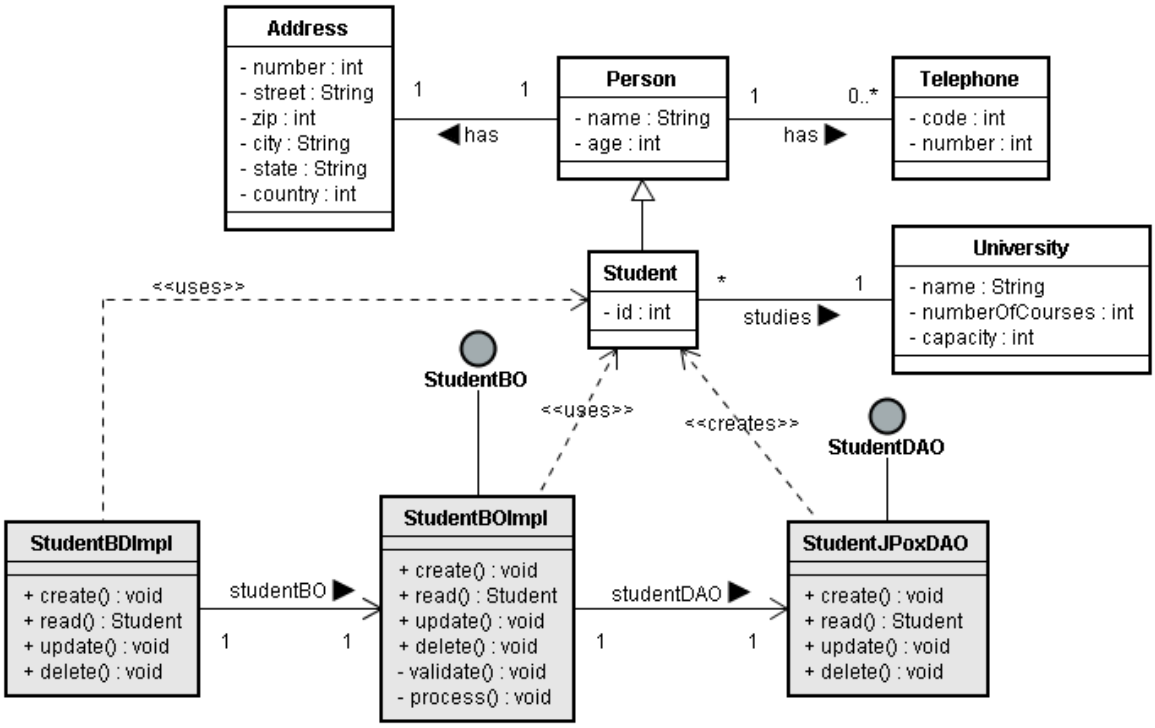


Figura 15. Diagrama de classes da aplicação após a geração

A **Figura 16** exibe um fragmento de código da classe *StudentBDImpl* que descreve como é obtido uma instância remota de *StudentBOImpl* utilizando o *framework Spring*.

```
private StudentBO studentBO;
public StudentBD() throws Exception {
    BeanFactoryLocator beanFactoryLocator = SingletonBeanFactoryLocator.getInstance("spring/beanRefFactory.xml");
    BeanFactory beanFactory = beanFactoryLocator.useBeanFactory("br.ufc.mcc.students").getFactory();
    this.studentBO = (StudentBOImpl) beanFactory.getBean("studentBO");
}
```

Figura 16. Obtenção de um BO

Do mesmo modo (**Figura 17**), a classe *StudentBOImpl* obtém uma instância de *StudentJPoxDAO* para fazer o gerenciamento da camada de persistência.

```
private StudentDAO studentDAO;
public StudentBOImpl() throws Exception {
    BeanFactoryLocator beanFactoryLocator = SingletonBeanFactoryLocator.getInstance("spring/beanRefFactory.xml");
    BeanFactory beanFactory = beanFactoryLocator.useBeanFactory("br.ufc.mcc.students").getFactory();
    this.studentDAO = (StudentJPoxDAO) beanFactory.getBean("studentDAO");
}
```

Figura 17. Obtenção de um DAO

Por fim, a **Figura 18** exibe o formato do código da *StudentJPoxDAO* responsável por fazer a persistência transparente dos objetos da aplicação.

```
public class StudentJPoxDAO extends org.springframework.orm.jdo.support.JdoDaoSupport {

    public void create(Student student) {
        getJdoTemplate().makePersistent(student);
    }

    (...)
}
```

Figura 18. Persistência transparente com JPox

6. Conclusões e Trabalhos Futuros

Ao fim deste trabalho conclui-se que a utilização de padrões no desenvolvimento de aplicações, independentemente do domínio de atuação, é um processo que requer dos desenvolvedores um elevado grau de conhecimento específico sobre onde encontrar os padrões, e como e quando utilizá-los. Além disso, observa-se que a utilização de padrões no desenvolvimento de aplicações distribuídas implica na realização de tarefas repetitivas e enfadonhas de codificação. Nesse contexto, a ferramenta apresentada neste artigo visa facilitar e difundir a utilização de padrões com o intuito de maximizar tanto a produtividade no desenvolvimento de *software* quanto a qualidade do *software* gerado.

Como trabalhos futuros, pretende-se fazer a interligação da ferramenta com repositórios de padrões para facilitar o processo de identificação de padrões relacionados para o desenvolvimento de aplicações para um domínio específico. Neste sentido, deseja-se expandir ao máximo o escopo de atuação da ferramenta a fim de possibilitar o desenvolvimento de aplicações para um número maior de domínios.

Referências

[1] Alur, D., Crupi, J., Malks, D. Core J2EE Patterns: Best Practices and Design Strategies. 2ed Edition. Prentice Hall, 2003.

[2] Alves, V. Progressive development of distributed object-oriented applications. Master's thesis, Centro de Informatica Universidade Federal de Pernambuco, Feb, 2001.

[3] Alves, V., Borba, P. An Implementation Method for Distributed Object-Oriented Applications. In *Second Latin American Conference on Pattern Languages of Programming*, SugarLoafPLoP'2002, pages 55-86, Itaipava, Brazil, 5th-7th August 2002.

[4] AndroMDA Tool. http://www.andromda.org. Acesso em janeiro de 2005.

[5] Apache. Velocity Template Engine (Velocity). http://jakarta.apache.org/velocity. Acesso em novembro de 2004.

[6] Bauer, C. and King, G. Hibernate in Action. Softbound, 2004.

[7] Booch, G., Rumbaugh, J. and Jacobson, I. The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[8] Dias, K., Borba, P. Padrões de projeto para estruturação de aplicações distribuídas Enterprise JavaBeans. In *Second Latin American Conference on Pattern Languages of Programming*, SugarLoafPLoP'2002, pages 55-86, Itaipava, Brazil, 5th-7th August 2002.

[9] Gentleware. Poseidon for UML. http://www.gentleware.com/index.php. Acesso em janeiro de 2005.

[10] Greve, F.G.P., Araújo, J.G.R., Andrade, S.S., Maia Filho, E.M.F., Brito, K.S., Rocha, L.A., Pinheiro, V.G. Cordel: uma Ferramenta Distribuída para a Geração de Aplicações Web. In *I2TS 3rd International Information and Telecommunication Technologies Symposium*, São Carlos, 2004. I2TS 3rd International Infor.

[11] Hibernate. Relational Persistence For Idiomatic Java. http://www.hibernate.org. Acesso em janeiro de 2005.

[12] IBM. Rational Rose. http://www-306.ibm.com/software/rational. Acesso em dezembro de 2004.

[13] Jpox - JDO. Java Persistent Object (JPOX).http://www.jpox.org/index.jsp. Acesso em janeiro de 2005.

[14] OMG. CORBA Specification. http://www.corba.org/. Acesso em julho de 2005.

[15] OMG. XML Metadata Interchange (XMI). www.omg.org/technology/documents/formal/xmi.htm. Acesso em dezembro de 2003.

[16] Reese, G. Database Programming with JDBC and Java. 2nd Edition. O'Reilly, 2000.

[17] Santana, E., Bianchini, C.P., Prado, A.F., Trevelin, L.C. Um Padrão para o Desenvolvimento de Software Baseado em Componentes Distribuídos. In Second Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'2002, pages 175-190, Itaipava, Brazil, 5th-7th August 2002.

[18] Schmidt, D.C., Fayad, M.E., Johnson, R.E. Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley Computing Publisher, 1999.

[19] Spring. Java/J2EE Application Framework. http://www.springframework.org. Acesso em janeiro de 2005.

[20] Sun Microsystems. Java Technology. http://java.sun.com, Acesso em janeiro de 2000.

[21] Sun Microsystems. Java 2 Platform, Enterprise Edition (J2EE). java.sun.com/j2ee, Acesso em dezembro de 2004.

[22] Sun Microsystems. Java Database Connectivity (JDBC). http://java.sun.com/products/jdbc. Acesso em julho de 2004.

[23] Sun Microsystems. Enterprise JavaBeans Technology (EJB). http://java.sun.com/products/ejb. Acesso em julho de 2004.

[24] Sun Microsystems. Remote Method Invocation (RMI). http://java.sun.com/products/jdk/rmi. Acesso em julho de 2005.

[25] Tigris. Modelling tool ArgoUML. http://argouml.tigris.org. Acesso em janeiro de 2005.

[26] Walls, C., Breidenbach, R. Spring in Action. Softbound, 2005.